EBook Gratis

APRENDIZAJE
Swift Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#swift

Tabla de contenido

Acerca de ... 1

Capítulo 1: Empezando con Swift Language ... 2

Observaciones ... 2

Otros recursos ... 2

Versiones ... 2

Examples ... 3

Tu primer programa Swift ... 3

Instalación de Swift ... 4

Tu primer programa en Swift en una Mac (usando un área de juegos) ... 5

Tu primer programa en la aplicación Swift Playgrounds para iPad ... 9

Valor opcional y enumeración opcional ... 12

Capítulo 2: (Inseguro) punteros de búfer ... 14

Introducción ... 14

Observaciones ... 14

Examples ... 14

UnsafeMutablePointer ... 15

Práctico caso de uso para punteros de búfer ... 16

Capítulo 3: Actuación ... 17

Examples ... 17

Asignación de rendimiento ... 17

Advertencia sobre estructuras con cadenas y propiedades que son clases ... 17

Capítulo 4: Algoritmos con Swift ... 19

Introducción ... 19

Examples ... 19

Tipo de inserción ... 19

Clasificación ... 20

Selección de selección ... 23

Análisis asintótico ... 23

Ordenación rápida: tiempo de complejidad O (n log n) ... 24

Gráfico, trie, pila ... 25

Grafico ... 25
Trie ... 33
Apilar ... 36
Capítulo 5: Almacenamiento en caché en el espacio en disco ... 39
Introducción ... 39
Examples ... 39
Ahorro ... 39
Leyendo ... 39
Capítulo 6: Arrays ... 40
Introducción ... 40
Sintaxis ... 40
Observaciones ... 40
Examples ... 40
Semántica de valor ... 40
Conceptos básicos de matrices ... 40
Arreglos vacíos ... 41
Literales array ... 41
Arreglos con valores repetidos. ... 41
Creando arrays desde otras secuencias. ... 41
Matrices multidimensionales ... 42
Acceso a valores de matriz ... 42
Metodos utiles ... 43
Modificar valores en una matriz ... 43
Ordenar una matriz ... 44
Creando una nueva matriz ordenada ... 44
Ordenar una matriz existente en su lugar ... 44
Ordenar una matriz con un orden personalizado ... 44
Transformando los elementos de un Array con mapa (_ :) ... 45
Extraer valores de un tipo dado de un Array con flatMap (_ :) ... 46
Filtrando una matriz ... 46
Filtrado nil de una transformación de Array con flatMap (_ :) ... 47

Suscribiendo una matriz con un rango 47
Agrupar valores de matriz 48
Aplanar el resultado de una transformación de Array con flatMap (_ :) 48
Combinando los caracteres en una serie de cadenas. 49
Aplanando una matriz multidimensional 49
Ordenar una matriz de cuerdas 50
Lácilmente aplanando una matriz multidimensional con aplanar () 51
Combinando los elementos de un Array con reduce (_: combine :) 51
Eliminar elemento de una matriz sin saber su índice 52
Swift3 52
Encontrar el elemento mínimo o máximo de un Array 52
Encontrar el elemento mínimo o máximo con un pedido personalizado. 53
Acceder a los índices de forma segura. 53
Comparando 2 matrices con zip 54
Capítulo 7: Bloques 55
Introducción 55
Examples 55
Cierre sin escape 55
Cierre de escape 55
Capítulo 8: Booleanos 57
Examples 57
¿Qué es Bool? 57
Niega un Bool con el prefijo! operador 57
Operadores lógicos booleanos 57
Booleanos y condicionales en línea 58
Capítulo 9: Bucles 60
Sintaxis 60
Examples 60
Bucle de entrada 60
Iterando sobre un rango 60
Iterando sobre una matriz o conjunto 60
Iterando sobre un diccionario 61

Iterando a la inversa....61
Iterando sobre rangos con zancada personalizada....62
Bucle de repetición....63
mientras bucle....63
Tipo de secuencia para cada bloque....63
Bucle de entrada con filtrado....64
Rompiendo un bucle....65
Capítulo 10: Cambiar....66
Parámetros....66
Observaciones....66
Examples....66
Uso básico....66
Coincidencia de valores múltiples....67
Haciendo juego un rango....67
Usando la instrucción where en un switch....67
Satisfacer una de las múltiples restricciones usando el interruptor....68
Emparejamiento parcial....68
Cambiar fracasos....69
Interruptor y Enums....70
Interruptor y Opcionales....70
Interruptores y tuplas....70
Coincidencia basada en clase - ideal para prepareForSegue....71
Capítulo 11: Cierres....73
Sintaxis....73
Observaciones....73
Examples....73
Fundamentos de cierre....73
Variaciones de sintaxis....74
Pasando cierres a funciones....75
Sintaxis de cierre de seguimiento....75
Parámetros de @noescape....75
Nota de Swift 3:....76

throws y rethrows ... 76
Capturas, referencias fuertes / débiles y ciclos de retención. ... 77
Retener ciclos ... 78
Utilización de cierres para codificación asíncrona. ... 78
Cierres y alias de tipo ... 79
Capítulo 12: Cifrado AES ... 81
Examples ... 81
Cifrado AES en modo CBC con un IV aleatorio (Swift 3.0) ... 81
Cifrado AES en modo CBC con un IV aleatorio (Swift 2.3) ... 84
Cifrado AES en modo ECB con relleno PKCS7 ... 85
Capítulo 13: Comenzando con la Programación Orientada al Protocolo ... 87
Observaciones ... 87
Examples ... 87
Aprovechamiento de la programación orientada al protocolo para pruebas unitarias ... 87
Usando protocolos como tipos de primera clase. ... 88
Capítulo 14: Concurrencia ... 93
Sintaxis ... 93
Examples ... 93
Obtención de una cola de Grand Central Dispatch (GCD) ... 93
Ejecución de tareas en una cola de Grand Central Dispatch (GCD) ... 95
Loops concurrentes ... 96
Ejecutar tareas en un OperationQueue ... 97
Creación de operaciones de alto nivel ... 99
Capítulo 15: Condicionales ... 102
Introducción ... 102
Observaciones ... 102
Examples ... 102
Usando guardia ... 102
Condicionales básicos: declaraciones if ... 103
El operador lógico AND ... 103
El operador lógico O ... 104
El operador lógico NO ... 104

Encuadernación opcional y cláusulas "donde" 104
Operador ternario 105
Operador sin coalescencia 106
Capítulo 16: Conjuntos 107
Examples 107
Declarar Conjuntos 107
Modificar valores en un conjunto. 107
Comprobando si un conjunto contiene un valor 107
Realización de operaciones en sets. 107
Agregando valores de mi propio tipo a un conjunto 108
CountedSet 108
Capítulo 17: Control de acceso 110
Sintaxis 110
Observaciones 110
Examples 110
Ejemplo básico usando un Struct 110
Car.make (público) 111
Car.model (interno) 111
Car.otherName (fileprivate) 111
Car.fullName (privado) 111
Ejemplo de subclasificación 111
Getters and Setters Ejemplo 112
Capítulo 18: Controlador de finalización 113
Introducción 113
Examples 113
Controlador de finalización sin argumento de entrada 113
Controlador de finalización con argumento de entrada 113
Capítulo 19: Convenciones de estilo 115
Observaciones 115
Examples 115
Claro uso 115
Evitar la ambigüedad 115

Evitar la redundancia 115
Nombrando variables de acuerdo a su rol. 115
Alto acoplamiento entre nombre de protocolo y nombres de variable 116
Proporcionar detalles adicionales cuando se utilizan parámetros de tipo débil 116
Uso fluido 116
Usando lenguaje natural 116
Métodos de fábrica de nombres 116
Nombrando Parámetros en Inicializadores y Métodos de Fábrica 116
Nombrar de acuerdo a los efectos secundarios 117
Funciones o variables booleanas 117
Protocolos de denominación 117
Tipos y propiedades 117
Capitalización 118
Tipos y Protocolos 118
Todo lo demas... 118
El caso de Carmel: 118
Abreviaturas 118
Capítulo 20: Cuerdas y personajes 120
Sintaxis 120
Observaciones 120
Examples 120
Literales de cuerdas y personajes 120
Interpolación de cuerdas 120
Caracteres especiales 121
Cuerdas de concatenación 121
Examina y compara cuerdas 122
Codificación y descomposición de cadenas 123
Cuerdas en descomposición 123
Longitud de la cuerda y la iteración 123
Unicode 123

Establecienda valores 123

Conversiones 124

Cuerdas de inversión 124

Cadenas mayúsculas y minúsculas 125

Compruebe si la cadena contiene caracteres de un conjunto definido 125

Contar las ocurrencias de un personaje en una cadena 126

Eliminar caracteres de una cadena no definida en Set 127

Formato de cadenas 127

Ceros a la izquierda 127

Números despues de decimal 127

Decimal a hexadecimal 127

Decimal a un número con radix arbitrario 127

Convertir una cadena Swift a un tipo de número 128

Iteración de cuerdas 128

Eliminar WhiteSpace y NewLine iniciales y finales 130

Convertir cadena a / desde datos / NSData 130

Dividir una cadena en una matriz 131

Capítulo 21: Derivación de clave PBKDF2 133

Examples 133

Clave basada en contraseña Derivación 2 (Swift 3) 133

Clave basada en contraseña Derivación 2 (Swift 2.3) 134

Calibración de derivación de clave basada en contraseña (Swift 2.3) 135

Calibración de derivación de clave basada en contraseña (Swift 3) 135

Capítulo 22: Entrar en Swift 137

Observaciones 137

Examples 137

Imprimir Debug 137

Actualización de una clase de depuración e impresión de valores. 137

tugurio 138

imprimir () vs dump () 139

imprimir vs NSLog 139

Capítulo 23: Enums 141
Observaciones 141
Examples 141
Enumeraciones basicas 141
Enums con valores asociados. 142
Cabida indirecta 143
Valores crudos y hash 143
Inicializadores 144
Las enumeraciones comparten muchas características con clases y estructuras 145
Enumeraciones anidadas 146
Capítulo 24: Estructuras 147
Examples 147
Fundamentos de Estructuras 147
Las estructuras son tipos de valor 147
Mutando un Struct 147
Cuando puedes usar métodos de mutación 148
Cuando NO puedes usar métodos de mutación 148
Las estructuras no pueden heredar 148
Accediendo miembros de struct 148
Capítulo 25: Extensiones 150
Observaciones 150
Examples 150
Variables y funciones 150
Inicializadores en extensiones 150
¿Qué son las extensiones? 151
Extensiones de protocolo 151
Restricciones 151
Qué son las extensiones y cuándo usarlas. 152
Subíndices 152
Capítulo 26: Funcionar como ciudadanos de primera clase en Swift 154
Introducción 154
Examples 154

Asignando función a una variable 154
Pasar la función como un argumento a otra función, creando así una función de orden superi 155
Función como tipo de retorno de otra función. 155
Capítulo 27: Funciones 156
Examples 156
Uso básico 156
Funciones con parámetros 156
Valores de retorno 157
Errores de lanzamiento 157
Métodos 158
Métodos de instancia 158
Métodos de tipo 158
Parámetros de entrada 158
Sintaxis de cierre de seguimiento 159
Los operadores son funciones 159
Parámetros variables 159
Subíndices 160
Opciones de subíndices: 161
Funciones con cierres 161
Funciones de paso y retorno. 162
Tipos de funciones 162
Capítulo 28: Funciones Swift Advance 164
Introducción 164
Examples 164
Introducción con funciones avanzadas. 164
Aplanar matriz multidimensional 165
Capítulo 29: Generar UIImage de Iniciales desde String 166
Introducción 166
Examples 166
InicialesImageFactory 166
Capítulo 30: Genéricos 167
Observaciones 167

Examples....167
Restricción de tipos genéricos de marcadores de posición....167
Los fundamentos de los genéricos....168
Funciones genéricas....168
Tipos genéricos....168
Pasando por tipos genéricos....169
Nombre genérico de marcador de posición....169
Ejemplos genéricos de clase....169
Herencia genérica de clase....170
Usando Genéricos para Simplificar las Funciones de Arreglos....171
Use genéricos para mejorar la seguridad de tipo....171
Restricciones de tipo avanzado....172
Capítulo 31: Gestión de la memoria....174
Introducción....174
Observaciones....174
Cuándo usar la palabra clave débil:....174
Cuándo usar la palabra clave sin dueño:....174
Escollos....174
Examples....174
Ciclos de referencia y referencias débiles....175
Referencias débiles....175
Gestión de memoria manual....176
Capítulo 32: Gestor de paquetes Swift....177
Examples....177
Creación y uso de un simple paquete Swift....177
Capítulo 33: Hash criptográfico....179
Examples....179
MD2, MD4, MD5, SHA1, SHA224, SHA256, SHA384, SHA512 (Swift 3)....179
HMAC con MD5, SHA1, SHA224, SHA256, SHA384, SHA512 (Swift 3)....180
Capítulo 34: Inicializadores....183
Examples....183

Establecer valores de propiedad predeterminados 183
Personalización de inicialización con paramatizadores. 183
Conveniencia inic 184
Inicializador designado 186
Inicia conveniencia () 186
Convenience init (otherString: String) 186
Inicializador designado (llamará al inicializador designado de superclase) 186
Inicia conveniencia () 187
Iniciador Throwable 187
Capítulo 35: Inyección de dependencia 188
Examples 188
Inyección de dependencia con controladores de vista 188
Introducción a la inyección dependiente 188
Ejemplo sin DI 188
Ejemplo con inyección de dependencia 189
Tipos de inyección de dependencia 191
Ejemplo de configuración sin DI 192
Inyección de dependencia del inicializador 192
Propiedades Dependencia Inyección 193
Método de inyección de dependencia 193
Capítulo 36: La declaración diferida 194
Examples 194
Cuándo usar una declaración diferida 194
Cuando NO usar una declaración diferida 194
Capítulo 37: Las clases 196
Observaciones 196
Examples 196
Definiendo una clase 196
Semántica de referencia 196
Propiedades y metodos 197
Clases y herencia múltiple 198

deinit....198

Capítulo 38: Lectura y escritura JSON....199

Sintaxis....199

Examples....199

Serialización, codificación y decodificación JSON con Apple Foundation y Swift Standard Li....199

Leer json....199

Escribir json....199

Codificar y decodificar automáticamente....200

Codificar a datos JSON....201

Decodificar a partir de datos JSON....201

Codificación o decodificación exclusiva....201

Usando Nombres Clave Personalizados....201

SwiftyJSON....202

Freddy....203

Ejemplo de datos JSON....203

Deserialización de datos sin procesar....204

Deserializando modelos directamente....205

Serialización de datos en bruto....205

Serialización de modelos directamente....205

Flecha....205

JSON simple de análisis en objetos personalizados....207

JSON analizando Swift 3....208

Capítulo 39: Los diccionarios....211

Observaciones....211

Examples....211

Declarar Diccionarios....211

Modificar los diccionarios....211

Valores de acceso....212

Cambiar el valor del diccionario usando la clave....212

Obtener todas las claves en el diccionario....213

Fusionar dos diccionarios....213

Capítulo 40: Manejo de errores....214

Observaciones ... 214
Examples ... 214
Fundamentos de manejo de errores ... 214
Capturando diferentes tipos de error ... 215
Patrón de captura y cambio para el manejo explícito de errores ... 216
Deshabilitando la propagación de errores ... 217
Crear error personalizado con descripción localizada ... 217
Capítulo 41: Marca de documentación ... 219
Examples ... 219
Documentación de la clase ... 219
Estilos de documentación ... 219
Capítulo 42: Método Swizzling ... 224
Observaciones ... 224
Campo de golf ... 224
Examples ... 224
Ampliando UIViewController y Swizzling viewDidLoad ... 224
Fundamentos de Swift Swiftling ... 225
Fundamentos de Swizzling - Objective-C ... 226
Capítulo 43: NSRegularExpression en Swift ... 227
Observaciones ... 227
Examples ... 227
Extendiendo String para hacer patrones simples ... 227
Uso básico ... 228
Sustitución de subcadenas ... 228
Caracteres especiales ... 229
Validación ... 229
NSRegularExpression para validación de correo ... 230
Capítulo 44: Números ... 231
Examples ... 231
Tipos de números y literales. ... 231
Literales ... 231
Sintaxis literal entera ... 231

Sintaxis literal de punto flotante 231
Convertir un tipo numérico a otro 232
Convertir números a / desde cadenas 232
Redondeo 233
redondo **233**
hacer techo **233**
piso **233**
En t **234**
Notas **234**
Generación de números aleatorios 234
Notas **234**
Exposiciónción 235
Capítulo 45: Objetos asociados **236**
Examples 236
Propiedad, en una extensión de protocolo, lograda utilizando objeto asociado. 236
Capítulo 46: Opcionales **239**
Introducción 239
Sintaxis 239
Observaciones 239
Examples 239
Tipos de opcionales 239
Desenvolviendo un opcional 240
Operador Coalescente Nil 241
Encadenamiento opcional 241
Descripción general - ¿Por qué Optionals? 242
Capítulo 47: Operadores Avanzados **244**
Examples 244
Operadores personalizados 244
Sobrecarga + para Diccionarios 245
Operadores conmutativos 245
Operadores de Bitwise 246

Operadores de desbordamiento ... 247
Precedencia de los operadores Swift estándar ... 247
Capítulo 48: OptionSet ... 249
Examples ... 249
Protocolo OptionSet ... 249
Capítulo 49: Patrones de diseño - Creacionales ... 250
Introducción ... 250
Examples ... 250
Semifallo ... 250
Método de fábrica ... 250
Observador ... 251
Cadena de responsabilidad ... 252
Iterador ... 253
Patrón de constructor ... 254
Ejemplo: ... 255
Llevarlo mas alla: ... 257
Capítulo 50: Patrones de diseño - Estructurales ... 261
Introducción ... 261
Examples ... 261
Adaptador ... 261
Fachada ... 261
Capítulo 51: Programación Funcional en Swift ... 263
Examples ... 263
Extraer una lista de nombres de una lista de personas ... 263
Atravesando ... 263
Saliente ... 263
Filtración ... 264
Usando Filtro con Estructuras ... 265
Capítulo 52: Protocolos ... 267
Introducción ... 267
Observaciones ... 267
Examples ... 267

Conceptos básicos del protocolo ... 267
Acerca de los protocolos ... 267
Requisitos de tipo asociado ... 269
Patrón de delegado ... 271
Ampliación del protocolo para una clase específica conforme ... 272
Usando el protocolo RawRepresentable (Extensible Enum) ... 273
Protocolos de clase solamente ... 273
Semántica de referencia de protocolos de clase solamente. ... 274
Variables débiles del tipo de protocolo. ... 275
Implementando el protocolo Hashable ... 275
Capítulo 53: Reflexión ... 277
Sintaxis ... 277
Observaciones ... 277
Examples ... 277
Uso básico para espejo ... 277
Obtención de tipos y nombres de propiedades para una clase sin tener que instanciarla ... 278
Capítulo 54: RxSwift ... 281
Examples ... 281
Conceptos básicos de RxSwift ... 281
Creando observables ... 281
Desechando ... 282
Fijaciones ... 283
RxCocoa y ControlEvents ... 283
Capítulo 55: Servidor HTTP Swift de Kitura ... 286
Introducción ... 286
Examples ... 286
Hola aplicación mundial ... 286
Capítulo 56: Tipo de fundición ... 290
Sintaxis ... 290
Examples ... 290
Downcasting ... 290
Casting con interruptor ... 290

Upcasting ... 291
Ejemplo de uso de un downcast en un parámetro de función que involucra subclases ... 291
Tipo casting en Swift Language ... 291
Tipo de fundición ... 291
Downcasting ... 292
Conversión de cadena a Int & Float: - ... 292
Conversión de flotar a cadena ... 292
Entero a valor de cadena ... 292
Flotante a valor de cadena ... 293
Valor flotante opcional a la cadena ... 293
Cadena opcional a valor de Int ... 293
Disminuyendo los valores de JSON ... 293
Disminución de valores desde JSON opcional ... 293
Gestionar JSON Response con condiciones. ... 293
Gestionar respuesta nula con condición ... 294
Salida: Empty Dictionary ... 294
Capítulo 57: Tipografías ... 295
Examples ... 295
Tipografías para cierres con parámetros. ... 295
Tipografías para cierres vacíos. ... 295
tipografías para otros tipos ... 295
Capítulo 58: Trabajando con C y Objective-C ... 296
Observaciones ... 296
Examples ... 296
Usando clases Swift desde el código Objective-C ... 296
En el mismo modulo ... 296
En otro modulo ... 297
Usando las clases de Objective-C del código Swift ... 297
Puente de encabezados ... 297
Interfaz generada ... 298
Especifique un encabezado de puente para swiftc ... 299

Usa un mapa de módulo para importar encabezados C 299
Interoperación de grano fino entre Objective-C y Swift 300
Usa la biblioteca estándar de C 301
Capítulo 59: Tuplas 302
Introducción 302
Observaciones 302
Examples 302
¿Qué son las tuplas? 302
La descomposición en variables individuales. 303
Tuplas como el valor de retorno de las funciones 303
Usando una tipografía para nombrar tu tipo de tupla 303
Intercambiando valores 304
Ejemplo con 2 variables 304
Ejemplo con 4 variables 304
Tuplas como caso en el interruptor 304
Capítulo 60: Variables y propiedades 306
Observaciones 306
Examples 306
Creando una Variable 306
Conceptos básicos de propiedad 306
Propiedades almacenadas perezosas 307
Propiedades calculadas 307
Variables locales y globales 308
Tipo de propiedades 308
Observadores de la propiedad 309
Creditos 310

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: swift-language

It is an unofficial and free Swift Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Swift Language.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Swift Language

Observaciones

Swift es un lenguaje de programación de aplicaciones y sistemas desarrollado por Apple y distribuido como código abierto . Swift interactúa con las API Objective-C y Cocoa / Cocoa touch para los sistemas operativos macOS, iOS, tvOS y watchOS de Apple. Swift actualmente soporta macOS y Linux. Los esfuerzos de la comunidad están en progreso para soportar Android, Windows y otras plataformas.

El desarrollo rápido sucede en GitHub ; Las contribuciones se envían generalmente a través de solicitudes de extracción .

Los errores y otros problemas se rastrean en bugs.swift.org .

Las discusiones sobre el desarrollo, la evolución y el uso de Swift se llevan a cabo en las listas de correo de Swift .

Otros recursos

- Swift (lenguaje de programación) (Wikipedia)
- El lenguaje de programación Swift (en línea)
- Referencia de la biblioteca estándar de Swift (en línea)
- Pautas de diseño de API (en línea)
- Swift Programming Series (iBooks)
- ... y más en developer.apple.com .

Versiones

Versión rápida	Versión Xcode	Fecha de lanzamiento
el desarrollo comenzó (primer compromiso)	-	2010-07-17
1.0	Xcode 6	2014-06-02
1.1	Xcode 6.1	2014-10-16
1.2	Xcode 6.3	2015-02-09
2.0	Xcode 7	2015-06-08
2.1	Xcode 7.1	2015-09-23
debut en código abierto	-	2015-12-03

Versión rápida	Versión Xcode	Fecha de lanzamiento
2.2	Xcode 7.3	2016-03-21
2.3	Xcode 8	2016-09-13
3.0	Xcode 8	2016-09-13
3.1	Xcode 8.3	2017-03-27

Examples

Tu primer programa Swift

Escriba su código en un archivo llamado `hello.swift` :

```
print("Hello, world!")
```

- Para compilar y ejecutar un script en un solo paso, use `swift` desde el terminal (en un directorio donde se encuentra este archivo):

  Para iniciar un terminal, presione `CTRL` + `ALT` + `T` en *Linux* , o búsquelo en Launchpad en *macOS* . Para cambiar el directorio, ingrese `cd directory_name` (o `cd ..` para regresar)

```
$ swift hello.swift
Hello, world!
```

> Un **compilador** es un programa informático (o un conjunto de programas) que transforma el código fuente escrito en un lenguaje de programación (el idioma fuente) en otro lenguaje informático (el idioma objetivo), y este último a menudo tiene una forma binaria conocida como código objeto. (Wikipedia)

- Para compilar y ejecutar por separado, use `swiftc` :

```
$ swiftc hello.swift
```

Esto compilará su código en el archivo de `hello` . Para ejecutarlo, ingrese `./` , seguido de un nombre de archivo.

```
$ ./hello
Hello, world!
```

- O use el swift REPL (Read-Eval-Print-Loop), escribiendo `swift` desde la línea de comandos, luego ingrese su código en el intérprete:

Código:

```
func greet(name: String, surname: String) {
```

```
    print("Greetings \(name) \(surname)")
}

let myName = "Homer"
let mySurname = "Simpson"

greet(name: myName, surname: mySurname)
```

Vamos a romper este gran código en pedazos:

- `func greet(name: String, surname: String) { // function body }` : crea una *función* que toma un `name` y un `surname` .
- `print("Greetings \(name) \(surname)")` - Esto imprime en la consola "Saludos", luego el `name` , luego el `surname` . Básicamente, `\(` **`variable_name`** `)` imprime el valor de esa variable.
- `let myName = "Homer"` y `let mySurname = "Simpson"` : cree *constantes* (variables cuyo valor no puede cambiar) usando `let` con nombres: `myName` , `mySurname` y valores: `"Homer"` , `"Simpson"` respectivamente.
- `greet(name: myName, surname: mySurname)` : llama a una *función* que creamos anteriormente y nos proporciona los valores de las *constantes* `myName` , `mySurname` .

Ejecutándolo usando REPL:

```
$  swift
Welcome to Apple Swift. Type :help for assistance.
  1> func greet(name: String, surname: String) {
  2.     print("Greetings \(name) \(surname)")
  3. }
  4>
  5> let myName = "Homer"
myName: String = "Homer"
  6> let mySurname = "Simpson"
mySurname: String = "Simpson"
  7> greet(name: myName, surname: mySurname)
Greetings Homer Simpson
  8> ^D
```

Presione `CTRL` + `D` para salir de REPL.

Instalación de Swift

Primero, descargue el compilador y los componentes.

A continuación, agregue Swift a su camino. En macOS, la ubicación predeterminada para la cadena de herramientas descargable es / Library / Developer / Toolchains. Ejecuta el siguiente comando en la Terminal:

```
export PATH=/Library/Developer/Toolchains/swift-latest.xctoolchain/usr/bin:"${PATH}"
```

En Linux, necesitarás instalar clang:

```
$ sudo apt-get install clang
```

Si instaló la cadena de herramientas Swift en un directorio que no sea la raíz del sistema, deberá ejecutar el siguiente comando, utilizando la ruta real de su instalación de Swift:

```
$ export PATH=/path/to/Swift/usr/bin:"${PATH}"
```

Puedes verificar que tienes la versión actual de Swift ejecutando este comando:

```
$ swift --version
```

Tu primer programa en Swift en una Mac (usando un área de juegos)

Desde su Mac, descargue e instale Xcode desde la Mac App Store siguiendo este enlace .

Una vez completada la instalación, abra Xcode y seleccione **Comenzar con un patio de juegos** :

En el siguiente panel, puede darle un nombre a su Patio de juegos o puede dejarlo en `MyPlayground` y presionar **Siguiente** :

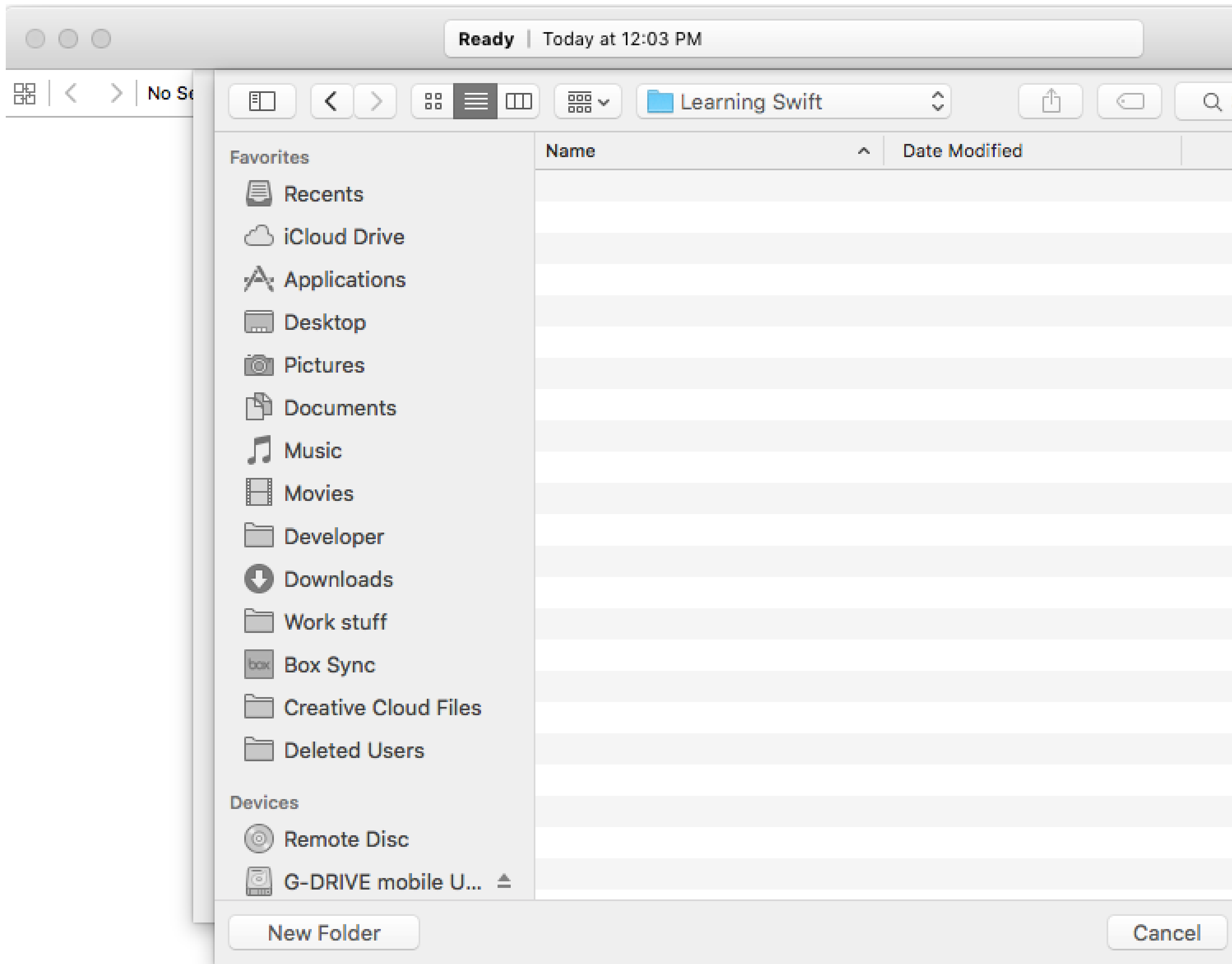
Choose options for your new playground:

Name MyPlayground

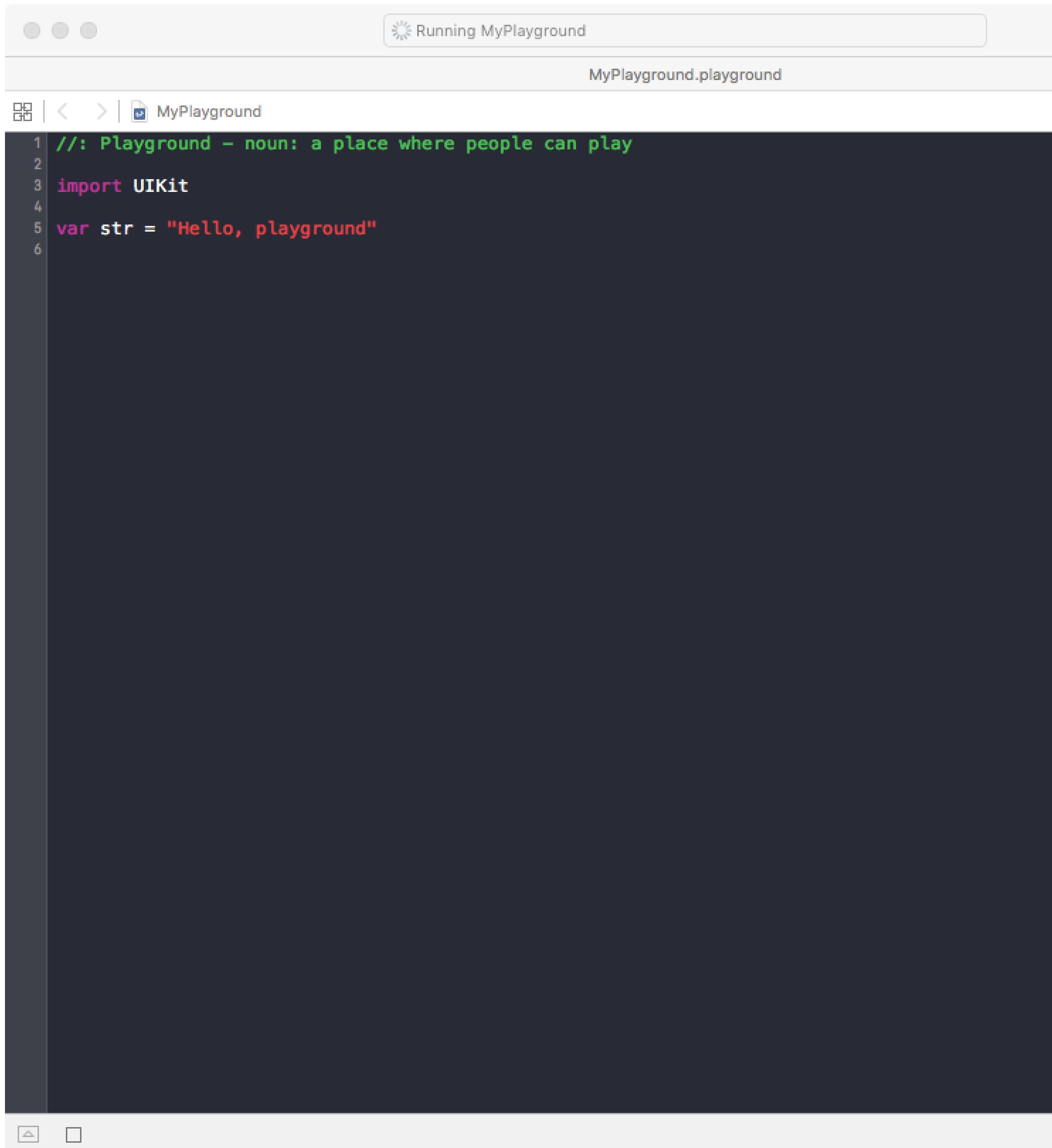
Platform: iOS

Cancel Previous Next

Seleccione una ubicación donde guardar el área de juegos y presione **Crear** :



El Patio de juegos se abrirá y su pantalla debería verse así:



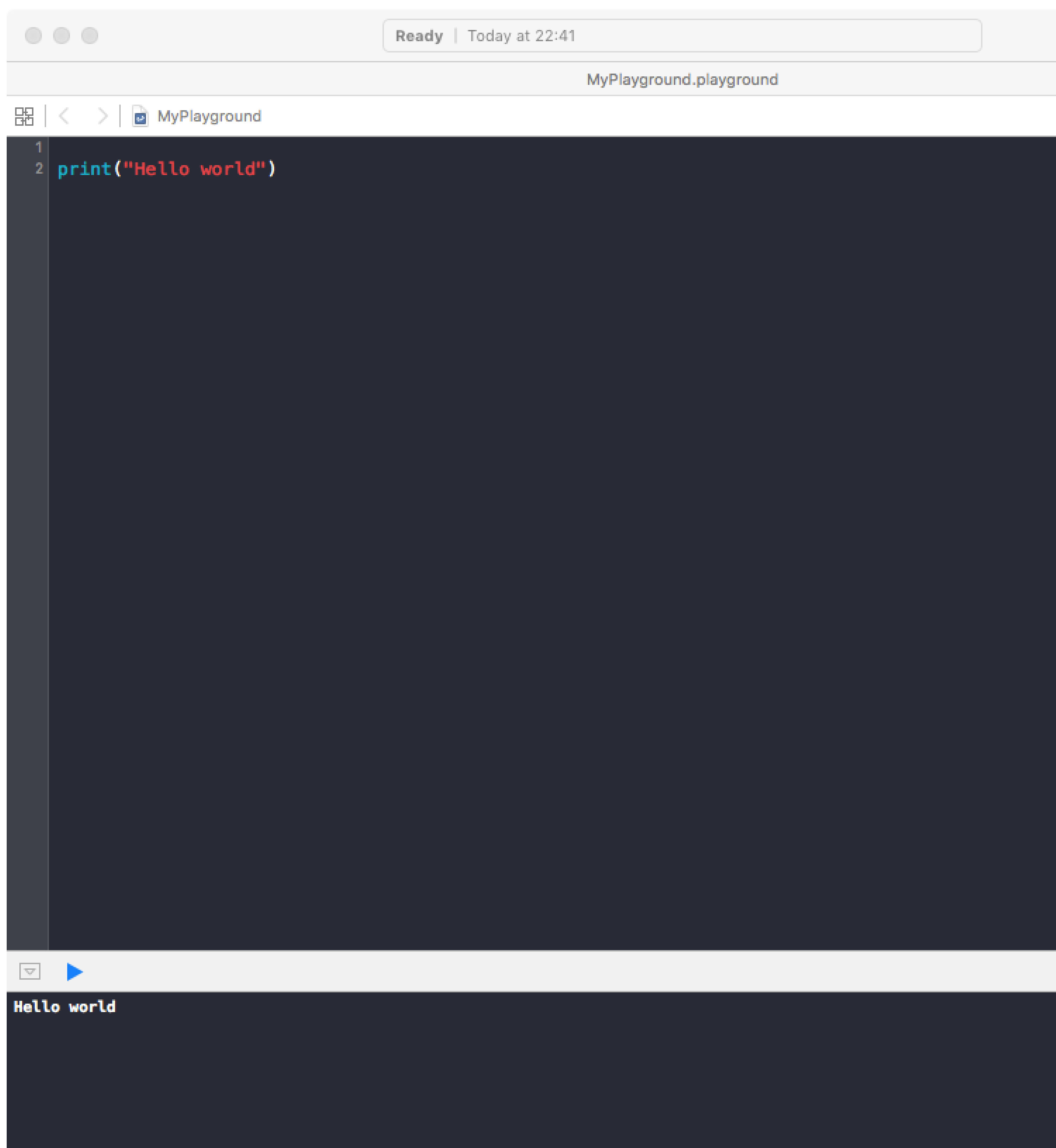
Ahora que el área de juegos está en la pantalla, presione ⇧ + cmd + Y para mostrar el **área de depuración** .

Finalmente borre el texto dentro de Playground y escriba:

```
print("Hello world")
```

Debería ver "Hola mundo" en el **Área de depuración** y "Hola mundo \ n" en la **barra lateral**

derecha:



¡Felicidades! ¡Has creado tu primer programa en Swift!

Tu primer programa en la aplicación Swift Playgrounds para iPad

La aplicación Swift Playgrounds es una excelente manera de comenzar a codificar Swift sobre la marcha. Para usarlo:

1- Descarga Swift Playgrounds para iPad desde la App Store.

2- Abre la aplicación

3- En la pestaña **Mis patios de juegos** , toque **+**

en la esquina superior izquierda y luego seleccione Plantilla en blanco.

4- Ingrese su código.

5- Toque Ejecutar mi código para ejecutar su código.

6- Al frente de cada línea, el resultado se almacenará en un pequeño cuadrado. Toque para revelar el resultado.

7- Para recorrer el código lentamente para rastrearlo, toque el botón junto a Ejecutar mi código.

Valor opcional y enumeración opcional

Tipo de opcionales, que maneja la ausencia de un valor. Los opcionales dicen que "hay un valor y es igual a x" o "no hay un valor en absoluto".

Un Opcional es un tipo en sí mismo, en realidad uno de los nuevos enums súper-potencia de Swift. Tiene dos valores posibles, `None` y `Some(T)` , donde T es un valor asociado del tipo de datos correcto disponible en Swift.

Echemos un vistazo a este fragmento de código, por ejemplo:

```
let x: String? = "Hello World"

if let y = x {
   print(y)
}
```

De hecho, si agrega una declaración de **`print(x.dynamicType)`** en el código anterior, verá esto en la consola:

```
Optional<String>
```

¿Cuerda? es en realidad azúcar sintáctica para Opcional, y Opcional es un tipo en sí mismo.

Aquí hay una versión simplificada del encabezado de Opcional, que puede ver haciendo clic con el botón derecho en la palabra Opcional en su código de Xcode:

```
enum Optional<Wrapped> {

 /// The absence of a value.
 case none

 /// The presence of a value, stored as `Wrapped`.
 case some(Wrapped)
}
```

Opcional es en realidad una enumeración, definida en relación con un tipo genérico Wrapped. Tiene dos casos: *`.none`* para representar la ausencia de un valor, y *`.some`* para representar la presencia de un valor, que se almacena como su valor asociado de tipo Wrapped.

Déjame pasar por eso otra vez: `String?` no es una `String` sino una `Optional<String>` . El hecho de que `Optional` sea un tipo significa que tiene sus propios métodos, por ejemplo, `map` y `flatMap` .

Lea Empezando con Swift Language en línea:
https://riptutorial.com/es/swift/topic/202/empezando-con-swift-language

Capítulo 2: (Inseguro) punteros de búfer

Introducción

“Se utiliza un puntero de búfer para el acceso de bajo nivel a una región de memoria. Por ejemplo, puede usar un puntero de búfer para el procesamiento y la comunicación eficientes de datos entre aplicaciones y servicios ".

Extracto de: Apple Inc. "Uso de Swift con Cocoa y Objective-C (Edición Swift 3.1)". IBooks. https://itun.es/us/utTW7.l

Usted es responsable de manejar el ciclo de vida de cualquier memoria con la que trabaje a través de punteros de búfer, para evitar fugas o comportamientos indefinidos.

Observaciones

Conceptos estrechamente alineados **requeridos** para completar la comprensión de los BufferPointers (inseguros).

- MemoryLayout (*el diseño de memoria de un tipo, que describe su tamaño, zancada y alineación*).
- No administrado (*un tipo para propagar una referencia de objeto no administrado*).
- UnsafeBufferPointer (*Una interfaz de colección no propietaria a un búfer de elementos almacenados de forma contigua en la memoria*).
- UnsafeBufferPointerIterator (*un iterador para los elementos en el búfer al que hace referencia una instancia de UnsafeBufferPointer o UnsafeMutableBufferPointer*).
- UnsafeMutableBufferPointer (*Una interfaz de colección no propietaria a un búfer de elementos mutables almacenados de forma contigua en la memoria*) *.*
- UnsafeMutablePointer (*Un puntero para acceder y manipular datos de un tipo específico*) *.*
- UnsafeMutableRawBufferPointer (*Una interfaz de colección no propietaria mutable a los bytes en una región de memoria*) *.*
- UnsafeMutableRawBufferPointer.Iterator (*un iterador sobre los bytes vistos por un puntero de búfer en bruto*) *.*
- UnsafeMutableRawPointer (*Un puntero en bruto para acceder y manipular datos sin tipo*) *.*
- UnsafePointer (*Un puntero para acceder a datos de un tipo específico*) *.*
- UnsafeRawBufferPointer (*Una interfaz de colección no propia para los bytes en una región de memoria*) *.*
- UnsafeRawBufferPointer.Iterator (*un iterador sobre los bytes vistos por un puntero de búfer en bruto*) *.*
- UnsafeRawPointer (*Un puntero en bruto para acceder a datos sin tipo*) *.*

  (Fuente, Swiftdoc.org)

Examples

UnsafeMutablePointer

```
struct UnsafeMutablePointer<Pointee>
```

> Un puntero para acceder y manipular datos de un tipo específico.

Utiliza instancias del tipo UnsafeMutablePointer para acceder a datos de un tipo específico en la memoria. El tipo de datos al que puede acceder un puntero es el tipo Pointee del puntero. UnsafeMutablePointer no ofrece gestión automatizada de la memoria ni garantías de alineación. Usted es responsable de manejar el ciclo de vida de cualquier memoria con la que trabaje mediante punteros inseguros para evitar fugas o comportamientos indefinidos.

La memoria que administras manualmente se puede desincronizar o enlazar a un tipo específico. Utiliza el tipo UnsafeMutablePointer para acceder y administrar la memoria que se ha vinculado a un tipo específico. (Fuente)

```
import Foundation

let arr = [1,5,7,8]

let pointer = UnsafeMutablePointer<[Int]>.allocate(capacity: 4)
pointer.initialize(to: arr)

let x = pointer.pointee[3]

print(x)

pointer.deinitialize()
pointer.deallocate(capacity: 4)

class A {
  var x: String?

  convenience init (_ x: String) {
    self.init()
    self.x = x
  }

  func description() -> String {
    return x ?? ""
  }
}


let arr2 = [A("OK"), A("OK 2")]
let pointer2 = UnsafeMutablePointer<[A]>.allocate(capacity: 2)
pointer2.initialize(to: arr2)

pointer2.pointee
let y = pointer2.pointee[1]

print(y)

pointer2.deinitialize()
pointer2.deallocate(capacity: 2)
```

Convertido a Swift 3.0 desde la fuente original

Práctico caso de uso para punteros de búfer

Deconstruir el uso de un puntero no seguro en el método de la biblioteca Swift;

```
public init?(validatingUTF8 cString: UnsafePointer<CChar>)
```

Propósito:

Crea una nueva cadena copiando y validando los datos UTF-8 terminados en nulo a los que hace referencia el puntero dado.

Este inicializador no intenta reparar secuencias de unidades de código UTF-8 mal formadas. Si se encuentra alguno, el resultado del inicializador es `nil` . El siguiente ejemplo llama a este inicializador con punteros al contenido de dos matrices `CChar` diferentes `CChar` el primero con secuencias de unidades de código UTF-8 bien formadas y el segundo con una secuencia mal formada al final.

> **Fuente** , **Apple Inc., archivo de encabezado Swift 3** (Para acceder al encabezado: en Playground, Cmd + Haga clic en la palabra Swift) en la línea de código:
>
> `import Swift`

```
let validUTF8: [CChar] = [67, 97, 102, -61, -87, 0]
     validUTF8.withUnsafeBufferPointer { ptr in
         let s = String(validatingUTF8: ptr.baseAddress!)
         print(s as Any)
     }
     // Prints "Optional(Café)"

     let invalidUTF8: [CChar] = [67, 97, 102, -61, 0]
     invalidUTF8.withUnsafeBufferPointer { ptr in
      let s = String(validatingUTF8: ptr.baseAddress!)
      print(s as Any)
     }
// Prints "nil"
```

> (Fuente, Apple Inc., Swift Header File Example)

Lea (Inseguro) punteros de búfer en línea: https://riptutorial.com/es/swift/topic/9140/-inseguro--punteros-de-bufer

Capítulo 3: Actuación

Examples

Asignación de rendimiento

En Swift, la administración de la memoria se realiza automáticamente mediante el conteo automático de referencias. (Consulte Administración de la memoria) La asignación es el proceso de reservar un lugar en la memoria para un objeto, y en la comprensión rápida de su rendimiento requiere cierta comprensión del **montón** y la **pila** . El montón es una ubicación de memoria donde se colocan la mayoría de los objetos, y usted puede pensar que es un cobertizo de almacenamiento. La pila, por otro lado, es una pila de llamadas de funciones que han conducido a la ejecución actual. (Por lo tanto, un seguimiento de pila es una especie de impresión de las funciones en la pila de llamadas).

Asignar y desasignar desde la pila es una operación muy eficiente, sin embargo, en comparación, la asignación del montón es costosa. Al diseñar para el rendimiento, debe tener esto en cuenta.

Clases:

```
class MyClass {

    let myProperty: String

}
```

Las clases en Swift son tipos de referencia y, por lo tanto, suceden varias cosas. Primero, el objeto real se asignará al montón. Entonces, cualquier referencia a ese objeto se debe agregar a la pila. Esto hace que las clases sean un objeto más caro para la asignación.

Estructuras:

```
struct MyStruct {

    let myProperty: Int

}
```

Como las estructuras son tipos de valor y, por lo tanto, se copian cuando se pasan, se asignan en la pila. Esto hace que las estructuras sean más eficientes que las clases; sin embargo, si necesita una noción de identidad y / o semántica de referencia, una estructura no puede proporcionarle esas cosas.

Advertencia sobre estructuras con cadenas y propiedades que son clases

Mientras que las estructuras son generalmente más limitadas que las clases, debe tener cuidado con las estructuras con propiedades que son clases:

```
struct MyStruct {

    let myProperty: MyClass

}
```

Aquí, debido al conteo de referencias y otros factores, el rendimiento ahora es más similar a una clase. Además, si más de una propiedad en la estructura es una clase, el impacto en el rendimiento puede ser incluso más negativo que si la estructura fuera una clase.

Además, si bien las cadenas son estructuras, almacenan internamente sus caracteres en el montón, por lo que son más caras que la mayoría de las estructuras.

Lea Actuación en línea: https://riptutorial.com/es/swift/topic/4067/actuacion

Capítulo 4: Algoritmos con Swift

Introducción

Los algoritmos son una columna vertebral de la computación. Al elegir qué algoritmo usar en qué situación se distingue un promedio de un buen programador. Con eso en mente, aquí hay definiciones y ejemplos de código de algunos de los algoritmos básicos que hay.

Examples

Tipo de inserción

El tipo de inserción es uno de los algoritmos más básicos en informática. La ordenación por inserción clasifica los elementos mediante la iteración a través de una colección y coloca los elementos en función de su valor. El conjunto se divide en mitades ordenadas y no clasificadas y se repite hasta que se ordenan todos los elementos. La ordenación por inserción tiene una complejidad de O (n2). Puede ponerlo en una extensión, como en un ejemplo a continuación, o puede crear un método para ello.

```
extension Array where Element: Comparable {

func insertionSort() -> Array<Element> {

    //check for trivial case
    guard self.count > 1 else {
        return self
    }

    //mutated copy
    var output: Array<Element> = self

    for primaryindex in 0..<output.count {

        let key = output[primaryindex]
        var secondaryindex = primaryindex

        while secondaryindex > -1 {
            if key < output[secondaryindex] {

                //move to correct position
                output.remove(at: secondaryindex + 1)
                output.insert(key, at: secondaryindex)
            }
            secondaryindex -= 1
        }
    }

    return output
}
}
```

Clasificación

Ordenamiento de burbuja

Este es un algoritmo de clasificación simple que recorre repetidamente la lista para ser ordenado, compara cada par de elementos adyacentes y los intercambia si están en el orden incorrecto. El paso por la lista se repite hasta que no se necesiten swaps. Aunque el algoritmo es simple, es demasiado lento y poco práctico para la mayoría de los problemas. Tiene una complejidad de O (n2) pero se considera más lenta que la ordenación por inserción.

```
extension Array where Element: Comparable {

func bubbleSort() -> Array<Element> {

    //check for trivial case
    guard self.count > 1 else {
        return self
    }

    //mutated copy
    var output: Array<Element> = self

    for primaryIndex in 0..<self.count {
        let passes = (output.count - 1) - primaryIndex

        //"half-open" range operator
        for secondaryIndex in 0..<passes {
            let key = output[secondaryIndex]

            //compare / swap positions
            if (key > output[secondaryIndex + 1]) {
                swap(&output[secondaryIndex], &output[secondaryIndex + 1])
            }
        }
    }

    return output
}

}
```

Tipo de inserción

El tipo de inserción es uno de los algoritmos más básicos en informática. La ordenación por inserción clasifica los elementos mediante la iteración a través de una colección y coloca los elementos en función de su valor. El conjunto se divide en mitades ordenadas y no clasificadas y se repite hasta que se ordenan todos los elementos. La ordenación por inserción tiene una complejidad de O (n2). Puede ponerlo en una extensión, como en un ejemplo a continuación, o puede crear un método para ello.

```
extension Array where Element: Comparable {

func insertionSort() -> Array<Element> {

    //check for trivial case
```

```
    guard self.count > 1 else {
        return self
    }

    //mutated copy
    var output: Array<Element> = self

    for primaryindex in 0..<output.count {

        let key = output[primaryindex]
        var secondaryindex = primaryindex

        while secondaryindex > -1 {
            if key < output[secondaryindex] {

                //move to correct position
                output.remove(at: secondaryindex + 1)
                output.insert(key, at: secondaryindex)
            }
            secondaryindex -= 1
        }
    }

    return output
}
}
```

Selección de selección

La selección por selección se caracteriza por su simplicidad. Comienza con el primer elemento de la matriz, guardando su valor como un valor mínimo (o máximo, dependiendo del orden de clasificación). Luego se itera a través de la matriz y reemplaza el valor mínimo por cualquier otro valor menor que el mínimo que encuentre en el camino. Ese valor mínimo se coloca en la parte más a la izquierda de la matriz y el proceso se repite, desde el siguiente índice, hasta el final de la matriz. La clasificación de selección tiene una complejidad de O (n2) pero se considera más lenta que su contraparte: la clasificación de selección.

func selectionSort () -> Array {// verifica si hay un caso de guardia de caso trivial> 1 else {return self}

```
//mutated copy
var output: Array<Element> = self

for primaryindex in 0..<output.count {
    var minimum = primaryindex
    var secondaryindex = primaryindex + 1

    while secondaryindex < output.count {
        //store lowest value as minimum
        if output[minimum] > output[secondaryindex] {
            minimum = secondaryindex
        }
        secondaryindex += 1
    }

    //swap minimum value with array iteration
    if primaryindex != minimum {
```

```
        swap(&output[primaryindex], &output[minimum])
    }
}

return output
}
```

Ordenación rápida: tiempo de complejidad O (n log n)

Quicksort es uno de los algoritmos avanzados. Cuenta con una complejidad de tiempo de O (n log n) y aplica una estrategia de dividir y conquistar. Esta combinación da como resultado un rendimiento algorítmico avanzado. Quicksort primero divide una gran matriz en dos sub-matrices más pequeñas: los elementos bajos y los elementos altos. Quicksort luego puede ordenar recursivamente los subarreglos.

Los pasos son:

Elija un elemento, llamado pivote, de la matriz.

Reordene la matriz para que todos los elementos con valores menores que el pivote aparezcan antes que el pivote, mientras que todos los elementos con valores mayores que el pivote aparezcan después (los valores iguales pueden ir en cualquier dirección). Después de esta partición, el pivote está en su posición final. Esto se llama la operación de partición.

Aplique recursivamente los pasos anteriores a la sub-matriz de elementos con valores más pequeños y por separado a la sub-matriz de elementos con valores mayores.

función mutante quickSort () -> Array {

```
func qSort(start startIndex: Int, _ pivot: Int) {

    if (startIndex < pivot) {
        let iPivot = qPartition(start: startIndex, pivot)
        qSort(start: startIndex, iPivot - 1)
        qSort(start: iPivot + 1, pivot)
    }
}
qSort(start: 0, self.endIndex - 1)
return self
}

mutating func qPartition(start startIndex: Int, _ pivot: Int) -> Int {

var wallIndex: Int = startIndex

//compare range with pivot
for currentIndex in wallIndex..<pivot {

    if self[currentIndex] <= self[pivot] {
        if wallIndex != currentIndex {
            swap(&self[currentIndex], &self[wallIndex])
        }

        //advance wall
        wallIndex += 1
```

```
        }
    }
    //move pivot to final position
    if wallIndex != pivot {
        swap(&self[wallIndex], &self[pivot])
    }
    return wallIndex
}
```

Selección de selección

La selección por selección se caracteriza por su simplicidad. Comienza con el primer elemento de la matriz, guardando su valor como un valor mínimo (o máximo, dependiendo del orden de clasificación). Luego se itera a través de la matriz y reemplaza el valor mínimo por cualquier otro valor menor que el mínimo que encuentre en el camino. Ese valor mínimo se coloca en la parte más a la izquierda de la matriz y el proceso se repite, desde el siguiente índice, hasta el final de la matriz. La clasificación de selección tiene una complejidad de O (n2) pero se considera más lenta que su contraparte: la clasificación de selección.

```
func selectionSort() -> Array<Element> {
    //check for trivial case
    guard self.count > 1 else {
        return self
    }

    //mutated copy
    var output: Array<Element> = self

    for primaryindex in 0..<output.count {
        var minimum = primaryindex
        var secondaryindex = primaryindex + 1

        while secondaryindex < output.count {
            //store lowest value as minimum
            if output[minimum] > output[secondaryindex] {
                minimum = secondaryindex
            }
            secondaryindex += 1
        }

        //swap minimum value with array iteration
        if primaryindex != minimum {
            swap(&output[primaryindex], &output[minimum])
        }
    }

    return output
}
```

Análisis asintótico

Dado que tenemos muchos algoritmos diferentes para elegir, cuando queremos ordenar una matriz, necesitamos saber cuál hará su trabajo. Así que necesitamos algún método para medir la velocidad y confiabilidad de algoritm. Ahí es donde comienza el análisis asintótico. El análisis asintótico es el proceso de describir la eficiencia de los algoritmos a medida que crece el tamaño

de entrada (n). En ciencias de la computación, los asintóticos generalmente se expresan en un formato común conocido como Big O Notation.

- **Tiempo lineal O (n)** : cuando cada elemento de la matriz debe evaluarse para que una función alcance su objetivo, eso significa que la función se vuelve menos eficiente a medida que aumenta la cantidad de elementos. *Se dice que una función como esta se ejecuta en tiempo lineal porque su velocidad depende de su tamaño de entrada.*
- **Tiempo polinominal O (n2)** : si la complejidad de una función crece exponencialmente (lo que significa que para n elementos de una matriz la complejidad de una función es n al cuadrado), esa función opera en tiempo polinominal. Estas suelen ser funciones con bucles anidados. Dos bucles anidados dan como resultado una complejidad O (n2), tres bucles anidados dan como resultado una complejidad O (n3), y así sucesivamente ...
- **Tiempo logarítmico O (log n):** la complejidad de las funciones de tiempo logarítmico se minimiza cuando el tamaño de sus entradas (n) aumenta. Estos son el tipo de funciones por las que se esfuerza todo programador.

Ordenación rápida: tiempo de complejidad O (n log n)

Quicksort es uno de los algoritmos avanzados. Cuenta con una complejidad de tiempo de O (n log n) y aplica una estrategia de dividir y conquistar. Esta combinación da como resultado un rendimiento algorítmico avanzado. Quicksort primero divide una gran matriz en dos sub-matrices más pequeñas: los elementos bajos y los elementos altos. Quicksort luego puede ordenar recursivamente los subarreglos.

Los pasos son:

1. Elija un elemento, llamado pivote, de la matriz.

2. Reordene la matriz para que todos los elementos con valores menores que el pivote aparezcan antes que el pivote, mientras que todos los elementos con valores mayores que el pivote aparezcan después (los valores iguales pueden ir en cualquier dirección). Después de esta partición, el pivote está en su posición final. Esto se llama la operación de partición.

3. Aplique recursivamente los pasos anteriores a la sub-matriz de elementos con valores más pequeños y por separado a la sub-matriz de elementos con valores mayores.

```
mutating func quickSort() -> Array<Element> {

func qSort(start startIndex: Int, _ pivot: Int) {

    if (startIndex < pivot) {
        let iPivot = qPartition(start: startIndex, pivot)
        qSort(start: startIndex, iPivot - 1)
        qSort(start: iPivot + 1, pivot)
    }
}
qSort(start: 0, self.endIndex - 1)
return self
```

}

función de mutación qPartición (iniciar índice de inicio: Int, _ pivote: Int) -> Int {

```
var wallIndex: Int = startIndex

//compare range with pivot
for currentIndex in wallIndex..<pivot {

    if self[currentIndex] <= self[pivot] {
        if wallIndex != currentIndex {
            swap(&self[currentIndex], &self[wallIndex])
        }

        //advance wall
        wallIndex += 1
    }
}
```

```
    //move pivot to final position
    if wallIndex != pivot {
        swap(&self[wallIndex], &self[pivot])
    }
    return wallIndex
}
```

Gráfico, trie, pila

Grafico

En ciencias de la computación, un gráfico es un tipo de datos abstractos destinado a implementar el gráfico no dirigido y los conceptos de gráficos dirigidos de las matemáticas. Una estructura de datos de gráfico consta de un conjunto finito (y posiblemente mutable) de vértices o nodos o puntos, junto con un conjunto de pares desordenados de estos vértices para un gráfico no dirigido o un conjunto de pares ordenados para un gráfico dirigido. Estos pares se conocen como bordes, arcos o líneas para un gráfico no dirigido y como flechas, bordes dirigidos, arcos dirigidos o líneas dirigidas para un gráfico dirigido. Los vértices pueden formar parte de la estructura del gráfico o pueden ser entidades externas representadas por índices o referencias de números enteros. Una estructura de datos de gráfico también puede asociar a cada borde algún valor de borde, como una etiqueta simbólica o un atributo numérico (costo, capacidad, longitud, etc.). (Wikipedia, fuente)

```
//
//  GraphFactory.swift
//  SwiftStructures
//
//  Created by Wayne Bishop on 6/7/14.
//  Copyright (c) 2014 Arbutus Software Inc. All rights reserved.
//
import Foundation


public class SwiftGraph {
```

```
    //declare a default directed graph canvas
    private var canvas: Array<Vertex>
    public var isDirected: Bool


    init() {
        canvas = Array<Vertex>()
        isDirected = true
    }


    //create a new vertex
    func addVertex(key: String) -> Vertex {


        //set the key
        let childVertex: Vertex = Vertex()
        childVertex.key = key


        //add the vertex to the graph canvas
        canvas.append(childVertex)


        return childVertex
    }



    //add edge to source vertex
    func addEdge(source: Vertex, neighbor: Vertex, weight: Int) {


        //create a new edge
        let newEdge = Edge()


        //establish the default properties
        newEdge.neighbor = neighbor
        newEdge.weight = weight
        source.neighbors.append(newEdge)


        print("The neighbor of vertex: \(source.key as String!) is \(neighbor.key as
String!)..")


        //check condition for an undirected graph
        if isDirected == false {


           //create a new reversed edge
           let reverseEdge = Edge()


           //establish the reversed properties
           reverseEdge.neighbor = source
           reverseEdge.weight = weight
           neighbor.neighbors.append(reverseEdge)

           print("The neighbor of vertex: \(neighbor.key as String!) is \(source.key as
```

```
String!)..")

        }


    }





    /* reverse the sequence of paths given the shortest path.
       process analagous to reversing a linked list. */

    func reversePath(_ head: Path!, source: Vertex) -> Path! {


        guard head != nil else {
            return head
        }

        //mutated copy
        var output = head


        var current: Path! = output
        var prev: Path!
        var next: Path!


        while(current != nil) {
            next = current.previous
            current.previous = prev
            prev = current
            current = next
        }


        //append the source path to the sequence
        let sourcePath: Path = Path()

        sourcePath.destination = source
        sourcePath.previous = prev
        sourcePath.total = nil

        output = sourcePath


        return output

    }





    //process Dijkstra's shortest path algorthim
    func processDijkstra(_ source: Vertex, destination: Vertex) -> Path? {


        var frontier: Array<Path> = Array<Path>()
        var finalPaths: Array<Path> = Array<Path>()
```

```
    //use source edges to create the frontier
    for e in source.neighbors {

        let newPath: Path = Path()


        newPath.destination = e.neighbor
        newPath.previous = nil
        newPath.total = e.weight


        //add the new path to the frontier
        frontier.append(newPath)

    }


    //construct the best path
    var bestPath: Path = Path()


    while frontier.count != 0 {

        //support path changes using the greedy approach
        bestPath = Path()
        var pathIndex: Int = 0


        for x in 0..<frontier.count {

            let itemPath: Path = frontier[x]

            if  (bestPath.total == nil) || (itemPath.total < bestPath.total) {
                bestPath = itemPath
                pathIndex = x
            }

        }



        //enumerate the bestPath edges
        for e in bestPath.destination.neighbors {

            let newPath: Path = Path()

            newPath.destination = e.neighbor
            newPath.previous = bestPath
            newPath.total = bestPath.total + e.weight


            //add the new path to the frontier
            frontier.append(newPath)

        }


        //preserve the bestPath
        finalPaths.append(bestPath)
```

```
        //remove the bestPath from the frontier
        //frontier.removeAtIndex(pathIndex) - Swift2
        frontier.remove(at: pathIndex)



    } //end while



    //establish the shortest path as an optional
    var shortestPath: Path! = Path()


    for itemPath in finalPaths {

        if (itemPath.destination.key == destination.key) {

            if  (shortestPath.total == nil) || (itemPath.total < shortestPath.total) {
                shortestPath = itemPath
            }

        }

    }


    return shortestPath

}



///an optimized version of Dijkstra's shortest path algorthim
func processDijkstraWithHeap(_ source: Vertex, destination: Vertex) -> Path! {


    let frontier: PathHeap = PathHeap()
    let finalPaths: PathHeap = PathHeap()


    //use source edges to create the frontier
    for e in source.neighbors {

        let newPath: Path = Path()


        newPath.destination = e.neighbor
        newPath.previous = nil
        newPath.total = e.weight


        //add the new path to the frontier
        frontier.enQueue(newPath)

    }


    //construct the best path
```

```
    var bestPath: Path = Path()


    while frontier.count != 0 {

        //use the greedy approach to obtain the best path
        bestPath = Path()
        bestPath = frontier.peek()


        //enumerate the bestPath edges
        for e in bestPath.destination.neighbors {

            let newPath: Path = Path()

            newPath.destination = e.neighbor
            newPath.previous = bestPath
            newPath.total = bestPath.total + e.weight


            //add the new path to the frontier
            frontier.enQueue(newPath)

        }


        //preserve the bestPaths that match destination
        if (bestPath.destination.key == destination.key) {
            finalPaths.enQueue(bestPath)
        }


        //remove the bestPath from the frontier
        frontier.deQueue()


    } //end while



    //obtain the shortest path from the heap
    var shortestPath: Path! = Path()
    shortestPath = finalPaths.peek()


    return shortestPath

}


//MARK: traversal algorithms


//bfs traversal with inout closure function
func traverse(_ startingv: Vertex, formula: (_ node: inout Vertex) -> ()) {


    //establish a new queue
    let graphQueue: Queue<Vertex> = Queue<Vertex>()
```

```
    //queue a starting vertex
    graphQueue.enQueue(startingv)


    while !graphQueue.isEmpty() {

        //traverse the next queued vertex
        var vitem: Vertex = graphQueue.deQueue() as Vertex!


        //add unvisited vertices to the queue
        for e in vitem.neighbors {
            if e.neighbor.visited == false {
                print("adding vertex: \(e.neighbor.key!) to queue..")
                graphQueue.enQueue(e.neighbor)
            }
        }


        /*
        notes: this demonstrates how to invoke a closure with an inout parameter.
        By passing by reference no return value is required.
        */

        //invoke formula
        formula(&vitem)


    } //end while


    print("graph traversal complete..")


}




//breadth first search
func traverse(_ startingv: Vertex) {


    //establish a new queue
    let graphQueue: Queue<Vertex> = Queue<Vertex>()


    //queue a starting vertex
    graphQueue.enQueue(startingv)


    while !graphQueue.isEmpty() {

        //traverse the next queued vertex
        let vitem = graphQueue.deQueue() as Vertex!

        guard vitem != nil else {
            return
        }

        //add unvisited vertices to the queue
```

```
        for e in vitem!.neighbors {
            if e.neighbor.visited == false {
                print("adding vertex: \(e.neighbor.key!) to queue..")
                graphQueue.enQueue(e.neighbor)
            }
        }


        vitem!.visited = true
        print("traversed vertex: \(vitem!.key!)..")


    } //end while


    print("graph traversal complete..")


} //end function



//use bfs with trailing closure to update all values
func update(startingv: Vertex, formula:((Vertex) -> Bool)) {


    //establish a new queue
    let graphQueue: Queue<Vertex> = Queue<Vertex>()


    //queue a starting vertex
    graphQueue.enQueue(startingv)


    while !graphQueue.isEmpty() {

        //traverse the next queued vertex
        let vitem = graphQueue.deQueue() as Vertex!

        guard vitem != nil else {
            return
        }

        //add unvisited vertices to the queue
        for e in vitem!.neighbors {
            if e.neighbor.visited == false {
                print("adding vertex: \(e.neighbor.key!) to queue..")
                graphQueue.enQueue(e.neighbor)
            }
        }


        //apply formula..
        if formula(vitem!) == false {
            print("formula unable to update: \(vitem!.key)")
        }
        else {
            print("traversed vertex: \(vitem!.key!)..")
        }

        vitem!.visited = true
```

```
        } //end while

        print("graph traversal complete..")

    }



}
```

Trie

En informática, un trie, también denominado árbol digital y, a veces, árbol de prefijo o árbol de prefijo (ya que pueden buscarse por prefijos), es un tipo de árbol de búsqueda: una estructura de datos de árbol ordenado que se utiliza para almacenar un conjunto dinámico o asociativo. Matriz donde las claves suelen ser cadenas. (Wikipedia, fuente)

```
//
//  Trie.swift
//  SwiftStructures
//
//  Created by Wayne Bishop on 10/14/14.
//  Copyright (c) 2014 Arbutus Software Inc. All rights reserved.
//
import Foundation


public class Trie {

    private var root: TrieNode!


    init(){
        root = TrieNode()
    }



    //builds a tree hierarchy of dictionary content
    func append(word keyword: String) {


        //trivial case
        guard keyword.length > 0 else {
            return
        }


        var current: TrieNode = root
```

```
        while keyword.length != current.level {

            var childToUse: TrieNode!
            let searchKey = keyword.substring(to: current.level + 1)


            //print("current has \(current.children.count) children..")


            //iterate through child nodes
            for child in current.children {

                if (child.key == searchKey) {
                    childToUse = child
                    break
                }

            }


            //new node
            if childToUse == nil {

                childToUse = TrieNode()
                childToUse.key = searchKey
                childToUse.level = current.level + 1
                current.children.append(childToUse)
            }


            current = childToUse


        } //end while


        //final end of word check
        if (keyword.length == current.level) {
            current.isFinal = true
            print("end of word reached!")
            return
        }



    } //end function




    //find words based on the prefix
    func search(forWord keyword: String) -> Array<String>! {


        //trivial case
        guard keyword.length > 0 else {
            return nil
        }


        var current: TrieNode = root
```

```
        var wordList = Array<String>()


        while keyword.length != current.level {

            var childToUse: TrieNode!
            let searchKey = keyword.substring(to: current.level + 1)


            //print("looking for prefix: \(searchKey)..")


            //iterate through any child nodes
            for child in current.children {

                if (child.key == searchKey) {
                    childToUse = child
                    current = childToUse
                    break
                }

            }


            if childToUse == nil {
               return nil
            }


        } //end while



        //retrieve the keyword and any descendants
        if ((current.key == keyword) && (current.isFinal)) {
            wordList.append(current.key)
        }


        //include only children that are words
        for child in current.children {

            if (child.isFinal == true) {
                wordList.append(child.key)
            }

        }


        return wordList


    } //end function


}
```

(GitHub, fuente)

Apilar

En informática, una pila es un tipo de datos abstracto que sirve como una colección de elementos, con dos operaciones principales: push, que agrega un elemento a la colección, y pop, que elimina el elemento agregado más reciente que aún no se eliminó. El orden en que los elementos salen de una pila da origen a su nombre alternativo, LIFO (para el último en entrar, el primero en salir). Además, una operación de inspección puede dar acceso a la parte superior sin modificar la pila. (Wikipedia, fuente)

Consulte la información de la licencia a continuación y la fuente del código original en (github)

```
//
//  Stack.swift
//  SwiftStructures
//
//  Created by Wayne Bishop on 8/1/14.
//  Copyright (c) 2014 Arbutus Software Inc. All rights reserved.
//
import Foundation


class Stack<T> {

    private var top: Node<T>

    init() {
        top = Node<T>()
    }


    //the number of items - O(n)
    var count: Int {


        //return trivial case
        guard top.key != nil else {
          return 0
        }


        var current = top
        var x: Int = 1


        //cycle through list
        while current.next != nil {
            current = current.next!
            x += 1
        }

        return x

    }


    //add item to the stack
    func push(withKey key: T) {
```

```
        //return trivial case
        guard top.key != nil else {
            top.key = key
            return
        }


        //create new item
        let childToUse = Node<T>()
        childToUse.key = key


        //set new created item at top
        childToUse.next = top
        top = childToUse

    }


    //remove item from the stack
    func pop() {

        if self.count > 1 {
            top = top.next
        }
        else {
            top.key = nil
        }

    }


    //retrieve the top most item
    func peek() -> T! {


        //determine instance
        if let topitem = top.key {
            return topitem
        }

        else {
            return nil
        }

    }



    //check for value
    func isEmpty() -> Bool {

        if self.count == 0 {
            return true
        }

        else {
            return false
        }
```

```
    }

}
```

La licencia MIT (MIT)

Copyright (c) 2015, Wayne Bishop & Arbutus Software Inc.

Por la presente, se otorga el permiso, sin cargo, a cualquier persona que obtenga una copia de este software y los archivos de documentación asociados (el "Software"), para operar el Software sin restricciones, incluidos, entre otros, los derechos de uso, copia, modificación, fusión. , publicar, distribuir, sublicenciar y / o vender copias del Software, y para permitir que las personas a quienes se suministra el Software lo hagan, sujeto a las siguientes condiciones:

El aviso de copyright anterior y este aviso de permiso se incluirán en todas las copias o partes sustanciales del Software.

EL SOFTWARE SE PROPORCIONA "TAL CUAL", SIN GARANTÍA DE NINGÚN TIPO, EXPRESO O IMPLÍCITO, INCLUYENDO PERO NO LIMITADO A LAS GARANTÍAS DE COMERCIABILIDAD, IDONEIDAD PARA UN PROPÓSITO PARTICULAR Y NO INCUMPLIMIENTO. EN NINGÚN CASO, LOS AUTORES O TITULARES DE DERECHOS DE AUTOR SERÁN RESPONSABLES POR CUALQUIER RECLAMACIÓN, DAÑOS U OTRAS RESPONSABILIDADES, YA QUE SEA RESPONSABLE DE UN CONTRATO, CORTE U OTRA MANERA, DERIVADOS DE, FUERA O EN CONEXIÓN CON EL SOFTWARE O EL USO U OTRAS REPARACIONES EN EL SOFTWARE.

Lea Algoritmos con Swift en línea: https://riptutorial.com/es/swift/topic/9116/algoritmos-con-swift

Capítulo 5: Almacenamiento en caché en el espacio en disco

Introducción

Vídeos de almacenamiento en caché, imágenes y audios utilizando `URLSession` y `FileManager`

Examples

Ahorro

```
let url = "https://path-to-media"
let request = URLRequest(url: url)
let downloadTask = URLSession.shared.downloadTask(with: request) { (location, response, error)
in
  guard let location = location,
        let response = response,
        let documentsPath = NSSearchPathForDirectoriesInDomains(.documentDirectory,
.userDomainMask, true).first else {
    return
  }
  let documentsDirectoryUrl = URL(fileURLWithPath: documentsPath)
  let documentUrl = documentsDirectoryUrl.appendingPathComponent(response.suggestedFilename)
  let _ = try? FileManager.default.moveItem(at: location, to: documentUrl)

  // documentUrl is the local URL which we just downloaded and saved to the FileManager
}.resume()
```

Leyendo

```
let url = "https://path-to-media"
guard let documentsUrl = FileManager.default.urls(for: .documentDirectory, in:
.userDomainMask).first,
      let searchQuery = url.absoluteString.components(separatedBy: "/").last else {
  return nil
}

do {
  let directoryContents = try FileManager.default.contentsOfDirectory(at: documentsUrl,
includingPropertiesForKeys: nil, options: [])
  let cachedFiles = directoryContents.filter { $0.absoluteString.contains(searchQuery) }

  // do something with the files found by the url
} catch {
  // Could not find any files
}
```

Lea Almacenamiento en caché en el espacio en disco en línea:
https://riptutorial.com/es/swift/topic/8902/almacenamiento-en-cache-en-el-espacio-en-disco

Capítulo 6: Arrays

Introducción

Array es un tipo de colección ordenada de acceso aleatorio. Las matrices son uno de los tipos de datos más utilizados en una aplicación. Usamos el tipo Array para contener elementos de un solo tipo, el tipo Element del array. Una matriz puede almacenar cualquier tipo de elementos, desde enteros hasta cadenas y clases.

Sintaxis

- Array <Element> // El tipo de una matriz con elementos de tipo Elemento
- [Elemento] // Azúcar sintáctico para el tipo de una matriz con elementos de tipo Elemento
- [element0, element1, element2, ... elementN] // Un literal de matriz
- [Elemento] () // Crea una nueva matriz vacía de tipo [Elemento]
- Array (count: repeatValue :) // Crea una matriz de elementos de `count` , cada uno inicializado en `repeatedValue`
- Array (_ :) // Crea un array a partir de una secuencia arbitraria

Observaciones

Las matrices son una *colección ordenada* de valores. Los valores pueden repetirse pero *deben* ser del mismo tipo.

Examples

Semántica de valor

Copiar una matriz copiará todos los elementos dentro de la matriz original.

Cambiar la nueva matriz *no cambiará* la matriz original.

```
var originalArray = ["Swift", "is", "great!"]
var newArray = originalArray
newArray[2] = "awesome!"
//originalArray = ["Swift", "is", "great!"]
//newArray = ["Swift", "is", "awesome!"]
```

Las matrices copiadas compartirán el mismo espacio en la memoria que el original hasta que se cambien. Como resultado de esto, se produce un impacto en el rendimiento cuando la matriz copiada recibe su propio espacio en la memoria cuando se modifica por primera vez.

Conceptos básicos de matrices

`Array` es un tipo de colección ordenada en la biblioteca estándar de Swift. Proporciona O (1)

acceso aleatorio y reasignación dinámica. La matriz es un tipo genérico , por lo que el tipo de valores que contiene se conocen en tiempo de compilación.

Dado que `Array` es un tipo de valor , su mutabilidad se define por su anotación como `var` (mutable) o `let` (immutable).

El tipo `[Int]` (que significa: una matriz que contiene `Int` s) es azúcar sintáctica para Array<T> .

Lee más sobre matrices en el lenguaje de programación Swift .

Arreglos vacíos

Las siguientes tres declaraciones son equivalentes:

```
// A mutable array of Strings, initially empty.

var arrayOfStrings: [String] = []      // type annotation + array literal
var arrayOfStrings = [String]()        // invoking the [String] initializer
var arrayOfStrings = Array<String>()   // without syntactic sugar
```

Literales array

Un literal de matriz se escribe con corchetes que rodean los elementos separados por comas:

```
// Create an immutable array of type [Int] containing 2, 4, and 7
let arrayOfInts = [2, 4, 7]
```

El compilador generalmente puede inferir el tipo de una matriz basada en los elementos en el literal, pero las **anotaciones de tipo** explícitas pueden invalidar el valor predeterminado:

```
let arrayOfUInt8s: [UInt8] = [2, 4, 7]   // type annotation on the variable
let arrayOfUInt8s = [2, 4, 7] as [UInt8] // type annotation on the initializer expression
let arrayOfUInt8s = [2 as UInt8, 4, 7]   // explicit for one element, inferred for the others
```

Arreglos con valores repetidos.

```
// An immutable array of type [String], containing ["Example", "Example", "Example"]
let arrayOfStrings = Array(repeating: "Example",count: 3)
```

Creando arrays desde otras secuencias.

```
let dictionary = ["foo" : 4, "bar" : 6]

// An immutable array of type [(String, Int)], containing [("bar", 6), ("foo", 4)]
```

```
let arrayOfKeyValuePairs = Array(dictionary)
```

Matrices multidimensionales

En Swift, se crea una matriz multidimensional anidando matrices: una matriz bidimensional de `Int` es `[[Int]]` (o `Array<Array<Int>>`).

```
let array2x3 = [
    [1, 2, 3],
    [4, 5, 6]
]
// array2x3[0][1] is 2, and array2x3[1][2] is 6.
```

Para crear una matriz multidimensional de valores repetidos, use llamadas anidadas del inicializador de matriz:

```
var array3x4x5 = Array(repeating: Array(repeating: Array(repeating: 0,count: 5),count:
4),count: 3)
```

Acceso a valores de matriz

Los siguientes ejemplos utilizarán esta matriz para demostrar el acceso a los valores.

```
var exampleArray:[Int] = [1,2,3,4,5]
//exampleArray = [1, 2, 3, 4, 5]
```

Para acceder a un valor en un índice conocido use la siguiente sintaxis:

```
let exampleOne = exampleArray[2]
//exampleOne = 3
```

Nota: El valor en el *índice dos es el tercer valor* en la `Array` . `Array` usa un *índice basado en cero, lo* que significa que el primer elemento de la `Array` está en el índice 0.

```
let value0 = exampleArray[0]
let value1 = exampleArray[1]
let value2 = exampleArray[2]
let value3 = exampleArray[3]
let value4 = exampleArray[4]
//value0 = 1
//value1 = 2
//value2 = 3
//value3 = 4
//value4 = 5
```

Acceda a un subconjunto de un `Array` utilizando el filtro:

```
var filteredArray = exampleArray.filter({ $0 < 4 })
//filteredArray = [1, 2, 3]
```

Los filtros pueden tener condiciones complejas como filtrar solo números pares:

```
var evenArray = exampleArray.filter({ $0 % 2 == 0 })
//evenArray = [2, 4]
```

También es posible devolver el índice de un valor dado, devolviendo `nil` si no se encontró el valor.

```
exampleArray.indexOf(3) // Optional(2)
```

Hay métodos para el primer, último, máximo o mínimo valor en una `Array` . Estos métodos devolverán `nil` si la `Array` está vacía.

```
exampleArray.first // Optional(1)
exampleArray.last // Optional(5)
exampleArray.maxElement() // Optional(5)
exampleArray.minElement() // Optional(1)
```

Metodos utiles

Determine si una matriz está vacía o devuelve su tamaño

```
var exampleArray = [1,2,3,4,5]
exampleArray.isEmpty //false
exampleArray.count //5
```

Invertir una matriz **Nota: el resultado no se realiza en la matriz en la que se invoca el método y se debe colocar en su propia variable.**

```
exampleArray = exampleArray.reverse()
//exampleArray = [9, 8, 7, 6, 5, 3, 2]
```

Modificar valores en una matriz

Hay varias formas de agregar valores a una matriz

```
var exampleArray = [1,2,3,4,5]
exampleArray.append(6)
//exampleArray = [1, 2, 3, 4, 5, 6]
var sixOnwards = [7,8,9,10]
exampleArray += sixOnwards
//exampleArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

y eliminar los valores de una matriz

```
exampleArray.removeAtIndex(3)
//exampleArray = [1, 2, 3, 5, 6, 7, 8, 9, 10]
exampleArray.removeLast()
//exampleArray = [1, 2, 3, 5, 6, 7, 8, 9]
exampleArray.removeFirst()
```

```
//exampleArray = [2, 3, 5, 6, 7, 8, 9]
```

Ordenar una matriz

```
var array = [3, 2, 1]
```

Creando una nueva matriz ordenada

Dado que `Array` ajusta a `SequenceType` , podemos generar una nueva matriz de los elementos ordenados utilizando un método de clasificación integrado.

2.1 2.2

En Swift 2, esto se hace con el método `sort()` .

```
let sorted = array.sort()  // [1, 2, 3]
```

3.0

A partir de Swift 3, ha sido renombrado a `sorted()` .

```
let sorted = array.sorted()  // [1, 2, 3]
```

Ordenar una matriz existente en su lugar

Como `Array` ajusta a `MutableCollectionType` , podemos ordenar sus elementos en su lugar.

2.1 2.2

En Swift 2, esto se hace usando el método `sortInPlace()` .

```
array.sortInPlace() // [1, 2, 3]
```

3.0

A partir de Swift 3, se le cambió el nombre a `sort()` .

```
array.sort() // [1, 2, 3]
```

> Nota: Para utilizar los métodos anteriores, los elementos deben cumplir con el protocolo `Comparable` .

Ordenar una matriz con un orden personalizado

También puede ordenar una matriz utilizando un cierre para definir si un elemento debe ordenarse antes que otro, lo cual no está restringido a matrices donde los elementos deben ser

`Comparable` . Por ejemplo, no tiene sentido que un `Landmark` de `Landmark` sea `Comparable` , pero aún puede ordenar un conjunto de puntos de referencia por altura o nombre.

```
struct Landmark {
    let name : String
    let metersTall : Int
}

var landmarks = [Landmark(name: "Empire State Building", metersTall: 443),
                 Landmark(name: "Eifell Tower", metersTall: 300),
                 Landmark(name: "The Shard", metersTall: 310)]
```

2.1 2.2

```
// sort landmarks by height (ascending)
landmarks.sortInPlace {$0.metersTall < $1.metersTall}

print(landmarks) // [Landmark(name: "Eifell Tower", metersTall: 300), Landmark(name: "The
Shard", metersTall: 310), Landmark(name: "Empire State Building", metersTall: 443)]

// create new array of landmarks sorted by name
let alphabeticalLandmarks = landmarks.sort {$0.name < $1.name}

print(alphabeticalLandmarks) // [Landmark(name: "Eifell Tower", metersTall: 300),
Landmark(name: "Empire State Building", metersTall: 443), Landmark(name: "The Shard",
metersTall: 310)]
```

3.0

```
// sort landmarks by height (ascending)
landmarks.sort {$0.metersTall < $1.metersTall}

// create new array of landmarks sorted by name
let alphabeticalLandmarks = landmarks.sorted {$0.name < $1.name}
```

> Nota: la comparación de cadenas puede producir resultados inesperados si las cadenas son inconsistentes, vea Ordenar una matriz de cadenas .

Transformando los elementos de un Array con mapa (_ :)

Como la `Array` ajusta al tipo de `SequenceType` , podemos usar `map(_:)` para transformar una matriz de `A` en una matriz de `B` utilizando un cierre de tipo `(A) throws -> B`

Por ejemplo, podríamos usarlo para transformar una matriz de `Int` s en una matriz de `String` s así:

```
let numbers = [1, 2, 3, 4, 5]
let words = numbers.map { String($0) }
print(words) // ["1", "2", "3", "4", "5"]
```

`map(_:)` recorrerá la matriz, aplicando el cierre dado a cada elemento. El resultado de ese cierre se utilizará para poblar una nueva matriz con los elementos transformados.

Como `String` tiene un inicializador que recibe un `Int` , también podemos usar esta sintaxis más clara:

```
let words = numbers.map(String.init)
```

Una transformación de `map(_:)` no necesita cambiar el tipo de la matriz; por ejemplo, también podría usarse para multiplicar una matriz de `Int` s por dos:

```
let numbers = [1, 2, 3, 4, 5]
let numbersTimes2 = numbers.map {$0 * 2}
print(numbersTimes2) // [2, 4, 6, 8, 10]
```

Extraer valores de un tipo dado de un Array con flatMap (_ :)

Las `things` Array contiene valores de `Any` tipo.

```
let things: [Any] = [1, "Hello", 2, true, false, "World", 3]
```

Podemos extraer valores de un tipo dado y crear una nueva matriz de ese tipo específico. Digamos que queremos extraer todos los `Int(s)` y colocarlos en un `Int` Array de forma segura.

```
let numbers = things.flatMap { $0 as? Int }
```

Ahora los `numbers` se definen como `[Int]` . La función `flatMap` descarta todos los elementos `nil` y el resultado contiene solo los siguientes valores:

```
[1, 2, 3]
```

Filtrando una matriz

Puede usar el método de `filter(_:)` en `SequenceType` para crear una nueva matriz que contenga los elementos de la secuencia que satisfagan un predicado dado, que se puede proporcionar como un cierre .

Por ejemplo, filtrando números pares de un `[Int]` :

```
let numbers = [22, 41, 23, 30]

let evenNumbers = numbers.filter { $0 % 2 == 0 }

print(evenNumbers)  // [22, 30]
```

Filtrando una `[Person]` , donde su edad es menor de 30 años:

```
struct Person {
    var age : Int
}

let people = [Person(age: 22), Person(age: 41), Person(age: 23), Person(age: 30)]

let peopleYoungerThan30 = people.filter { $0.age < 30 }

print(peopleYoungerThan30) // [Person(age: 22), Person(age: 23)]
```

Filtrado nil de una transformación de Array con flatMap (_ :)

Puede usar `flatMap(_:)` de una manera similar para `map(_:)` para crear una matriz aplicando una transformación a los elementos de una secuencia.

```
extension SequenceType {
    public func flatMap<T>(@noescape transform: (Self.Generator.Element) throws -> T?)
rethrows -> [T]
}
```

La diferencia con esta versión de `flatMap(_:)` es que espera que el cierre de transformación devuelva un valor Opcional `T?` Para cada uno de los elementos. Luego desenvolverá de forma segura cada uno de estos valores opcionales, filtrando `nil` , dando como resultado una matriz de `[T]` .

Por ejemplo, puede hacer esto para transformar un `[String]` en un `[Int]` usando `String` inicializador de `String` failable de `Int` , filtrando cualquier elemento que no pueda convertirse:

```
let strings = ["1", "foo", "3", "4", "bar", "6"]

let numbersThatCanBeConverted = strings.flatMap { Int($0) }

print(numbersThatCanBeConverted) // [1, 3, 4, 6]
```

También puede usar la `flatMap(_:)` para filtrar `nil` a fin de simplemente convertir una matriz de opcionales en una matriz de no opcionales:

```
let optionalNumbers : [Int?] = [nil, 1, nil, 2, nil, 3]

let numbers = optionalNumbers.flatMap { $0 }

print(numbers) // [1, 2, 3]
```

Suscribiendo una matriz con un rango

Uno puede extraer una serie de elementos consecutivos de un Array usando un Rango.

```
let words = ["Hey", "Hello", "Bonjour", "Welcome", "Hi", "Hola"]
let range = 2...4
let slice = words[range] // ["Bonjour", "Welcome", "Hi"]
```

La suscripción de una matriz con un rango devuelve un `ArraySlice` . Es una subsecuencia del Array.

En nuestro ejemplo, tenemos un Array of Strings, así que volvemos a `ArraySlice<String>` .

Aunque un ArraySlice se ajusta al `CollectionType` y se puede usar con `sort` , `filter` , etc., su propósito no es para el almacenamiento a largo plazo sino para los cálculos transitorios: debe volver a convertirse en un Array tan pronto como haya terminado de trabajar con él.

Para esto, use el inicializador `Array()` :

```
let result = Array(slice)
```

Para resumir en un ejemplo simple sin pasos intermedios:

```
let words = ["Hey", "Hello", "Bonjour", "Welcome", "Hi", "Hola"]
let selectedWords = Array(words[2...4]) // ["Bonjour", "Welcome", "Hi"]
```

Agrupar valores de matriz

Si tenemos una estructura como esta

```
struct Box {
    let name: String
    let thingsInside: Int
}
```

y una matriz de `Box(es)`

```
let boxes = [
    Box(name: "Box 0", thingsInside: 1),
    Box(name: "Box 1", thingsInside: 2),
    Box(name: "Box 2", thingsInside: 3),
    Box(name: "Box 3", thingsInside: 1),
    Box(name: "Box 4", thingsInside: 2),
    Box(name: "Box 5", thingsInside: 3),
    Box(name: "Box 6", thingsInside: 1)
]
```

podemos agrupar los cuadros por la propiedad `thingsInside` para obtener un `Dictionary` donde la `key` es el número de cosas y el valor es una matriz de cuadros.

```
let grouped = boxes.reduce([Int:[Box]]()) { (res, box) -> [Int:[Box]] in
    var res = res
    res[box.thingsInside] = (res[box.thingsInside] ?? []) + [box]
    return res
}
```

Ahora agrupado es un `[Int:[Box]]` y tiene el siguiente contenido

```
[
    2: [Box(name: "Box 1", thingsInside: 2), Box(name: "Box 4", thingsInside: 2)],
    3: [Box(name: "Box 2", thingsInside: 3), Box(name: "Box 5", thingsInside: 3)],
    1: [Box(name: "Box 0", thingsInside: 1), Box(name: "Box 3", thingsInside: 1), Box(name:
"Box 6", thingsInside: 1)]
]
```

Aplanar el resultado de una transformación de Array con flatMap (_ :)

Además de poder crear una matriz al filtrar `nil` de los elementos transformados de una secuencia,

también hay una versión de `flatMap(_:)` que espera que el cierre de la transformación devuelva una secuencia `S`

```
extension SequenceType {
    public func flatMap<S : SequenceType>(transform: (Self.Generator.Element) throws -> S)
rethrows -> [S.Generator.Element]
}
```

Cada secuencia de la transformación se concatenará, dando como resultado una matriz que contiene los elementos combinados de cada secuencia - `[S.Generator.Element]` .

Combinando los caracteres en una serie de cadenas.

Por ejemplo, podemos usarlo para tomar una matriz de cadenas principales y combinar sus caracteres en una sola matriz:

```
let primes = ["2", "3", "5", "7", "11", "13", "17", "19"]
let allCharacters = primes.flatMap { $0.characters }
// => "["2", "3", "5", "7", "1", "1", "1", "3", "1", "7", "1", "9"]"
```

Desglosando el ejemplo anterior:

1. `primes` es una `[String]` (Como una matriz es una secuencia, podemos llamar a `flatMap(_:)` en ella).
2. El cierre de la transformación toma uno de los elementos de los `primes` , una `String` (`Array<String>.Generator.Element`).
3. El cierre luego devuelve una secuencia de tipo `String.CharacterView` .
4. El resultado es una matriz que contiene los elementos combinados de todas las secuencias de cada una de las llamadas de cierre de transformación: `[String.CharacterView.Generator.Element]` .

Aplanando una matriz multidimensional

Como `flatMap(_:)` concatenará las secuencias devueltas de las llamadas de cierre de transformación, puede utilizarse para aplanar una matriz multidimensional, como una matriz 2D en una matriz 1D, una matriz 3D en una matriz 2D, etc.

Esto se puede hacer simplemente devolviendo el elemento dado `$0` (una matriz anidada) en el cierre:

```
// A 2D array of type [[Int]]
let array2D = [[1, 3], [4], [6, 8, 10], [11]]

// A 1D array of type [Int]
let flattenedArray = array2D.flatMap { $0 }

print(flattenedArray) // [1, 3, 4, 6, 8, 10, 11]
```

Ordenar una matriz de cuerdas

3.0

La forma más sencilla es usar `sorted()` :

```
let words = ["Hello", "Bonjour", "Salute", "Ahola"]
let sortedWords = words.sorted()
print(sortedWords) // ["Ahola", "Bonjour", "Hello", "Salute"]
```

o `sort()`

```
var mutableWords = ["Hello", "Bonjour", "Salute", "Ahola"]
mutableWords.sort()
print(mutableWords) // ["Ahola", "Bonjour", "Hello", "Salute"]
```

Puede pasar un cierre como argumento para clasificar:

```
let words = ["Hello", "Bonjour", "Salute", "Ahola"]
let sortedWords = words.sorted(isOrderedBefore: { $0 > $1 })
print(sortedWords) // ["Salute", "Hello", "Bonjour", "Ahola"]
```

Sintaxis alternativa con un cierre final:

```
let words = ["Hello", "Bonjour", "Salute", "Ahola"]
let sortedWords = words.sorted() { $0 > $1 }
print(sortedWords) // ["Salute", "Hello", "Bonjour", "Ahola"]
```

Pero habrá resultados inesperados si los elementos de la matriz no son consistentes:

```
let words = ["Hello", "bonjour", "Salute", "ahola"]
let unexpected = words.sorted()
print(unexpected) // ["Hello", "Salute", "ahola", "bonjour"]
```

Para resolver este problema, ordene en una versión minúscula de los elementos:

```
let words = ["Hello", "bonjour", "Salute", "ahola"]
let sortedWords = words.sorted { $0.lowercased() < $1.lowercased() }
print(sortedWords) // ["ahola", "bonjour", "Hello", "Salute"]
```

O `import Foundation` y use los métodos de comparación de `caseInsensitiveCompare` como `caseInsensitiveCompare` :

```
let words = ["Hello", "bonjour", "Salute", "ahola"]
let sortedWords = words.sorted { $0.caseInsensitiveCompare($1) == .orderedAscending }
print(sortedWords) // ["ahola", "bonjour", "Hello", "Salute"]
```

Alternativamente, use `localizedCaseInsensitiveCompare` , que puede administrar los signos diacríticos.

Para ordenar adecuadamente las cadenas por el valor *numérico* que contienen, use `compare` con la opción `.numeric` :

```
let files = ["File-42.txt", "File-01.txt", "File-5.txt", "File-007.txt", "File-10.txt"]
let sortedFiles = files.sorted() { $0.compare($1, options: .numeric) == .orderedAscending }
print(sortedFiles) // ["File-01.txt", "File-5.txt", "File-007.txt", "File-10.txt", "File-
42.txt"]
```

Lácilmente aplanando una matriz multidimensional con aplanar ()

Podemos usar `flatten()` para reducir perezosamente el anidamiento de una secuencia multidimensional.

Por ejemplo, laxitud aplanar una matriz 2D en una matriz 1D:

```
// A 2D array of type [[Int]]
let array2D = [[1, 3], [4], [6, 8, 10], [11]]

// A FlattenBidirectionalCollection<[[Int]]>
let lazilyFlattenedArray = array2D.flatten()

print(lazilyFlattenedArray.contains(4)) // true
```

En el ejemplo anterior, `flatten()` devolverá una `FlattenBidirectionalCollection` , que aplicará perezosamente el aplanamiento de la matriz. Por lo tanto, `contains(_:)` solo requerirá que las dos primeras matrices anidadas de `array2D` , ya que se producirá un cortocircuito al encontrar el elemento deseado.

Combinando los elementos de un Array con reduce (_: combine :)

`reduce(_:combine:)` se puede utilizar para combinar los elementos de una secuencia en un solo valor. Se requiere un valor inicial para el resultado, así como un cierre para aplicar a cada elemento, que devolverá el nuevo valor acumulado.

Por ejemplo, podemos usarlo para sumar una matriz de números:

```
let numbers = [2, 5, 7, 8, 10, 4]

let sum = numbers.reduce(0) {accumulator, element in
    return accumulator + element
}

print(sum) // 36
```

Pasamos `0` al valor inicial, ya que ese es el valor inicial lógico para una suma. Si pasamos en un valor de `N` , la `sum` resultante sería `N + 36` . El cierre aprobado para `reduce` tiene dos argumentos. `accumulator` es el valor acumulado actual, al cual se le asigna el valor que el cierre devuelve en cada iteración. `element` es el elemento actual en la iteración.

Como en este ejemplo, estamos pasando un cierre `(Int, Int) -> Int` para `reduce` , que simplemente está generando la adición de las dos entradas; de hecho, podemos pasar el

operador + directamente, ya que los operadores son funciones en Swift:

```
let sum = numbers.reduce(0, combine: +)
```

Eliminar elemento de una matriz sin saber su índice

Generalmente, si queremos eliminar un elemento de una matriz, necesitamos saber su índice para poder eliminarlo fácilmente usando la función `remove(at:)` .

¡Pero qué pasa si no conocemos el índice, pero sabemos el valor del elemento que se eliminará!

Así que aquí está la extensión simple a una matriz que nos permitirá eliminar un elemento de la matriz fácilmente sin saber su índice:

Swift3

```
extension Array where Element: Equatable {

    mutating func remove(_ element: Element) {
        _ = index(of: element).flatMap {
            self.remove(at: $0)
        }
    }
}
```

p.ej

```
    var array = ["abc", "lmn", "pqr", "stu", "xyz"]
    array.remove("lmn")
    print("\(array)")    //["abc", "pqr", "stu", "xyz"]

    array.remove("nonexistent")
    print("\(array)")    //["abc", "pqr", "stu", "xyz"]
    //if provided element value is not present, then it will do nothing!
```

También si, por error, hicimos algo como esto: `array.remove(25)` es decir, proporcionamos valor con diferentes tipos de datos, el compilador arrojará un error mencionando:
`cannot convert value to expected argument type`

Encontrar el elemento mínimo o máximo de un Array

2.1 2.2

Puede usar los `minElement()` y `maxElement()` para encontrar el elemento mínimo o máximo en una secuencia dada. Por ejemplo, con una matriz de números:

```
let numbers = [2, 6, 1, 25, 13, 7, 9]

let minimumNumber = numbers.minElement() // Optional(1)
let maximumNumber = numbers.maxElement() // Optional(25)
```

3.0

A partir de Swift 3, los métodos se han cambiado a `min()` y `max()` respectivamente:

```
let minimumNumber = numbers.min() // Optional(1)
let maximumNumber = numbers.max() // Optional(25)
```

Los valores devueltos de estos métodos son opcionales para reflejar el hecho de que la matriz podría estar vacía; si lo está, se devolverá `nil` .

> Nota: Los métodos anteriores requieren que los elementos se ajusten al protocolo `Comparable` .

Encontrar el elemento mínimo o máximo con un pedido personalizado.

También puede usar los métodos anteriores con un cierre personalizado, definiendo si un elemento debe ordenarse antes que otro, lo que le permite encontrar el elemento mínimo o máximo en una matriz donde los elementos no son necesariamente `Comparable` .

Por ejemplo, con una matriz de vectores:

```
struct Vector2 {
    let dx : Double
    let dy : Double

    var magnitude : Double {return sqrt(dx*dx+dy*dy)}
}

let vectors = [Vector2(dx: 3, dy: 2), Vector2(dx: 1, dy: 1), Vector2(dx: 2, dy: 2)]
```

2.1 2.2

```
// Vector2(dx: 1.0, dy: 1.0)
let lowestMagnitudeVec2 = vectors.minElement { $0.magnitude < $1.magnitude }

// Vector2(dx: 3.0, dy: 2.0)
let highestMagnitudeVec2 = vectors.maxElement { $0.magnitude < $1.magnitude }
```

3.0

```
let lowestMagnitudeVec2 = vectors.min { $0.magnitude < $1.magnitude }
let highestMagnitudeVec2 = vectors.max { $0.magnitude < $1.magnitude }
```

Acceder a los índices de forma segura.

Al agregar la siguiente extensión a la matriz, se puede acceder a los índices sin saber si el índice está dentro de los límites.

```
extension Array {
    subscript (safe index: Int) -> Element? {
        return indices ~= index ? self[index] : nil
```

```
    }
}
```

ejemplo:

```
if let thirdValue = array[safe: 2] {
    print(thirdValue)
}
```

Comparando 2 matrices con zip

La función `zip` acepta 2 parámetros de tipo `SequenceType` y devuelve una `Zip2Sequence` donde cada elemento contiene un valor de la primera secuencia y uno de la segunda secuencia.

Ejemplo

```
let nums = [1, 2, 3]
let animals = ["Dog", "Cat", "Tiger"]
let numsAndAnimals = zip(nums, animals)
```

nomsAndAnimals ahora contiene los siguientes valores

secuencia1	secuencia1
`1`	`"Dog"`
`2`	`"Cat"`
`3`	`"Tiger"`

Esto es útil cuando desea realizar algún tipo de comparación entre el elemento n-th de cada Array.

Ejemplo

Dados 2 matrices de `Int(s)`

```
let list0 = [0, 2, 4]
let list1 = [0, 4, 8]
```

desea verificar si cada valor en la `list1` es el doble del valor relacionado en la `list0` .

```
let list1HasDoubleOfList0 = !zip(list0, list1).filter { $0 != (2 * $1)}.isEmpty
```

Lea Arrays en línea: https://riptutorial.com/es/swift/topic/284/arrays

Capítulo 7: Bloques

Introducción

Desde Swift Documentarion

Se dice que un cierre escapa a una función cuando el cierre se pasa como un argumento a la función, pero se llama después de que la función retorna. Cuando declara una función que toma un cierre como uno de sus parámetros, puede escribir @escaping antes del tipo de parámetro para indicar que se permite que el cierre se escape.

Examples

Cierre sin escape

En Swift 1 y 2, los parámetros de cierre se escapaban por defecto. Si sabía que su cierre no escaparía al cuerpo de la función, podría marcar el parámetro con el atributo @noescape.

En Swift 3, es al revés: los parámetros de cierre no se escapan por defecto. Si pretende que se escape de la función, debe marcarla con el atributo @escaping.

```
class ClassOne {
  // @noescape is applied here as default
  func methodOne(completion: () -> Void) {
    //
  }
}

class ClassTwo {
  let obj = ClassOne()
  var greeting = "Hello, World!"

  func methodTwo() {
    obj.methodOne() {
      // self.greeting is required
      print(greeting)
    }
  }
}
```

Cierre de escape

Desde Swift Documentarion

> @escaping
>
> Aplique este atributo al tipo de parámetro en una declaración de método o función para indicar que el valor del parámetro se puede almacenar para su ejecución posterior. Esto significa que se permite que el valor supere la vida útil de la llamada.

> Los parámetros de tipo de función con el atributo de tipo de escape requieren el uso explícito de self. Para propiedades o métodos.

```
class ClassThree {

    var closure: (() -> ())?

    func doSomething(completion: @escaping () -> ()) {
        closure = finishBlock
    }
}
```

En el ejemplo anterior, el bloque de finalización se guarda en el cierre y, literalmente, vivirá más allá de la llamada a la función. Así que complier forzará a marcar el bloque de finalización como @escaping.

Lea Bloques en línea: https://riptutorial.com/es/swift/topic/8623/bloques

Capítulo 8: Booleanos

Examples

¿Qué es Bool?

Bool es un tipo booleano con dos valores posibles: `true` y `false` .

```
let aTrueBool = true
let aFalseBool = false
```

Bools se utilizan en las declaraciones de control de flujo como condiciones. La instrucción `if` usa una condición booleana para determinar qué bloque de código se ejecutará:

```
func test(_ someBoolean: Bool) {
    if someBoolean {
        print("IT'S TRUE!")
    }
    else {
        print("IT'S FALSE!")
    }
}
test(aTrueBool)  // prints "IT'S TRUE!"
```

Niega un Bool con el prefijo! operador

El prefijo `!` El operador devuelve la **negación lógica** de su argumento. Es decir `!true` devuelve `false` , y `!false` devuelve `true` .

```
print(!true)  // prints "false"
print(!false) // prints "true"

func test(_ someBoolean: Bool) {
    if !someBoolean {
        print("someBoolean is false")
    }
}
```

Operadores lógicos booleanos

El operador OR (||) devuelve verdadero si uno de sus dos operandos se evalúa como verdadero, de lo contrario, devuelve falso. Por ejemplo, el siguiente código se evalúa como verdadero porque al menos una de las expresiones a ambos lados del operador OR es verdadera:

```
if (10 < 20) || (20 < 10) {
    print("Expression is true")
}
```

El operador AND (&&) devuelve verdadero solo si ambos operandos evalúan ser verdaderos. El

siguiente ejemplo devolverá falso porque solo una de las dos expresiones de operando se evalúa como verdadera:

```
if (10 < 20) && (20 < 10) {
    print("Expression is true")
}
```

El operador XOR (^) devuelve verdadero si uno y solo uno de los dos operandos se evalúa como verdadero. Por ejemplo, el siguiente código devolverá verdadero ya que solo un operador evalúa ser verdadero:

```
if (10 < 20) ^ (20 < 10) {
    print("Expression is true")
}
```

Booleanos y condicionales en línea

Una forma limpia de manejar los booleanos es usar un condicional en línea con a? b: operador ternario, que forma parte de las Operaciones Básicas de Swift.

El condicional en línea se compone de 3 componentes:

```
question ? answerIfTrue : answerIfFalse
```

donde question es un valor booleano que se evalúa y answerIfTrue es el valor devuelto si la pregunta es verdadera, y answerIfFalse es el valor devuelto si la pregunta es falsa.

La expresión de arriba es la misma que:

```
if question {
    answerIfTrue
} else {
    answerIfFalse
}
```

Con los condicionales en línea, devuelve un valor basado en un valor booleano:

```
func isTurtle(_ value: Bool) {
    let color = value ? "green" : "red"
    print("The animal is \(color)")
}

isTurtle(true) // outputs 'The animal is green'
isTurtle(false) // outputs 'The animal is red'
```

También puede llamar a métodos basados en un valor booleano:

```
func actionDark() {
    print("Welcome to the dark side")
}
```

```
func actionJedi() {
    print("Welcome to the Jedi order")
}

func welcome(_ isJedi: Bool) {
    isJedi ? actionJedi() : actionDark()
}

welcome(true) // outputs 'Welcome to the Jedi order'
welcome(false) // outputs 'Welcome to the dark side'
```

Los condicionales en línea permiten realizar evaluaciones booleanas limpias de una línea

Lea Booleanos en línea: https://riptutorial.com/es/swift/topic/735/booleanos

Capítulo 9: Bucles

Sintaxis

- para constantes en secuencia {sentencias}
- para constante en secuencia donde condición {declaraciones}
- para var variable en secuencia {sentencias}
- para _ en secuencia {sentencias}
- para caso dejar constante en secuencia {sentencias}
- para el caso se deja constante en secuencia donde condición {declaraciones}
- para el caso var variable en secuencia {sentencias}
- condición {declaraciones}
- repite {sentencias} mientras condición
- sequence.forEach (body: (Element) throws -> Void)

Examples

Bucle de entrada

El bucle **for-in** te permite iterar sobre cualquier secuencia.

Iterando sobre un rango

Puede iterar en rangos medio abiertos y cerrados:

```
for i in 0..<3 {
    print(i)
}

for i in 0...2 {
    print(i)
}

// Both print:
// 0
// 1
// 2
```

Iterando sobre una matriz o conjunto

```
let names = ["James", "Emily", "Miles"]

for name in names {
   print(name)
}

// James
```

```
// Emily
// Miles
```

2.1 2.2

Si necesita el índice para cada elemento de la matriz, puede usar el método `enumerate()` en `SequenceType` .

```
for (index, name) in names.enumerate() {
   print("The index of \(name) is \(index).")
}

// The index of James is 0.
// The index of Emily is 1.
// The index of Miles is 2.
```

`enumerate()` devuelve una secuencia perezosa que contiene pares de elementos con `Int` s consecutivos, comenzando desde 0. Por lo tanto, con matrices, estos números corresponderán al índice dado de cada elemento; sin embargo, esto puede no ser el caso con otros tipos de colecciones.

3.0

En Swift 3, `enumerate()` ha sido renombrado a `enumerated()` :

```
for (index, name) in names.enumerated() {
   print("The index of \(name) is \(index).")
}
```

Iterando sobre un diccionario

```
let ages = ["James": 29, "Emily": 24]

for (name, age) in ages {
    print(name, "is", age, "years old.")
}

// Emily is 24 years old.
// James is 29 years old.
```

Iterando a la inversa

2.1 2.2

Puede usar el método `reverse()` en `SequenceType` para iterar sobre cualquier secuencia a la inversa:

```
for i in (0..<3).reverse() {
    print(i)
}

for i in (0...2).reverse() {
```

```
    print(i)
}

// Both print:
// 2
// 1
// 0

let names = ["James", "Emily", "Miles"]

for name in names.reverse() {
    print(name)
}

// Miles
// Emily
// James
```

3.0

En Swift 3, `reverse()` ha cambiado su nombre a `reversed()` :

```
for i in (0..<3).reversed() {
    print(i)
}
```

Iterando sobre rangos con zancada personalizada

2.1 2.2

Mediante el uso de los métodos `stride(_:_:)` en `Strideable` puede iterar en un rango con un paso personalizado:

```
for i in 4.stride(to: 0, by: -2) {
    print(i)
}

// 4
// 2

for i in 4.stride(through: 0, by: -2) {
    print(i)
}

// 4
// 2
// 0
```

1.2 3.0

En Swift 3, los métodos de `stride(_:_:)` en `Stridable` han sido reemplazados por las funciones de `stride(_:_:_:)` global `stride(_:_:_:)` :

```
for i in stride(from: 4, to: 0, by: -2) {
    print(i)
```

```
}

for i in stride(from: 4, through: 0, by: -2) {
    print(i)
}
```

Bucle de repetición

De manera similar al bucle while, solo se evalúa la instrucción de control después del bucle. Por lo tanto, el bucle siempre se ejecutará al menos una vez.

```
var i: Int = 0

repeat {
   print(i)
   i += 1
} while i < 3

// 0
// 1
// 2
```

mientras bucle

Un `while` de bucle se ejecutará siempre que la condición es verdadera.

```
var count = 1

while count < 10 {
    print("This is the \(count) run of the loop")
    count += 1
}
```

Tipo de secuencia para cada bloque

Un tipo que se ajuste al protocolo SequenceType puede recorrer sus elementos dentro de un cierre:

```
collection.forEach { print($0) }
```

Lo mismo podría hacerse con un parámetro nombrado:

```
collection.forEach { item in
    print(item)
}
```

* Nota: las declaraciones de flujo de control (como break o continue) no pueden usarse en estos bloques. Se puede llamar a una devolución y, si se llama, devolverá inmediatamente el bloque para la iteración actual (como lo haría una continuación). La siguiente iteración se ejecutará.

```
let arr = [1,2,3,4]
```

```
arr.forEach {

    // blocks for 3 and 4 will still be called
    if $0 == 2 {
        return
    }
}
```

Bucle de entrada con filtrado.

1. **`where`** cláusula

Al agregar una cláusula `where` , puede restringir las iteraciones a aquellas que satisfagan la condición dada.

```
for i in 0..<5 where i % 2 == 0 {
    print(i)
}

// 0
// 2
// 4


let names = ["James", "Emily", "Miles"]

for name in names where name.characters.contains("s") {
    print(name)
}

// James
// Miles
```

2. cláusula de **`case`**

Es útil cuando necesita iterar solo a través de los valores que coinciden con algún patrón:

```
let points = [(5, 0), (31, 0), (5, 31)]
for case (_, 0) in points {
    print("point on x-axis")
}

//point on x-axis
//point on x-axis
```

¿También puede filtrar valores opcionales y desenvolverlos si es apropiado agregando `?` marcar después de la constante de unión:

```
let optionalNumbers = [31, 5, nil]
for case let number? in optionalNumbers {
    print(number)
}

//31
```

```
//5
```

Rompiendo un bucle

Un bucle se ejecutará mientras su condición siga siendo verdadera, pero puede detenerlo manualmente usando la palabra clave `break` . Por ejemplo:

```
var peopleArray = ["John", "Nicole", "Thomas", "Richard", "Brian", "Novak", "Vick", "Amanda",
"Sonya"]
var positionOfNovak = 0

for person in peopleArray {
    if person == "Novak" { break }
    positionOfNovak += 1
}

print("Novak is the element located on position [\(positionOfNovak)] in peopleArray.")
//prints out: Novak is the element located on position 5 in peopleArray. (which is true)
```

Lea Bucles en línea: https://riptutorial.com/es/swift/topic/1186/bucles

Capítulo 10: Cambiar

Parámetros

Parámetro	Detalles
Valor a prueba	La variable que comparar contra

Observaciones

Proporcionar un caso para cada valor posible de su entrada. Utilice un `default case` para cubrir los valores de entrada restantes que no desea especificar. El caso por defecto debe ser el último caso.

Por defecto, Switches en Swift no continuará verificando otros casos después de que un caso haya sido comparado.

Examples

Uso básico

```
let number = 3
switch number {
case 1:
    print("One!")
case 2:
    print("Two!")
case 3:
    print("Three!")
default:
    print("Not One, Two or Three")
}
```

Las instrucciones de conmutación también funcionan con tipos de datos que no sean enteros. Funcionan con cualquier tipo de datos. Aquí hay un ejemplo de cómo cambiar una cadena:

```
let string = "Dog"
switch string {
case "Cat", "Dog":
  print("Animal is a house pet.")
default:
  print("Animal is not a house pet.")
}
```

Esto imprimirá lo siguiente:

```
Animal is a house pet.
```

Coincidencia de valores múltiples

Un solo caso en una instrucción de conmutación puede coincidir en varios valores.

```
let number = 3
switch number {
case 1, 2:
    print("One or Two!")
case 3:
    print("Three!")
case 4, 5, 6:
    print("Four, Five or Six!")
default:
    print("Not One, Two, Three, Four, Five or Six")
}
```

Haciendo juego un rango

Un solo caso en una instrucción de cambio puede coincidir con un rango de valores.

```
let number = 20
switch number {
case 0:
    print("Zero")
case 1..<10:
    print("Between One and Ten")
case 10..<20:
    print("Between Ten and Twenty")
case 20..<30:
    print("Between Twenty and Thirty")
default:
    print("Greater than Thirty or less than Zero")
}
```

Usando la instrucción where en un switch

La declaración donde se puede usar dentro de una coincidencia de caso de conmutador para agregar criterios adicionales requeridos para una coincidencia positiva. El siguiente ejemplo comprueba no solo el rango, sino también si el número es impar o par:

```
switch (temperature) {
      case 0...49 where temperature % 2 == 0:
        print("Cold and even")

      case 50...79 where temperature % 2 == 0:
        print("Warm and even")

      case 80...110 where temperature % 2 == 0:
        print("Hot and even")

      default:
        print("Temperature out of range or odd")
}
```

Satisfacer una de las múltiples restricciones usando el interruptor

Puedes crear una tupla y usar un interruptor como tal:

```
var str: String? = "hi"
var x: Int? = 5

switch (str, x) {
case (.Some,.Some):
    print("Both have values")
case (.Some, nil):
    print("String has a value")
case (nil, .Some):
    print("Int has a value")
case (nil, nil):
    print("Neither have values")
}
```

Emparejamiento parcial

Cambiar la instrucción hace uso de la coincidencia parcial.

```
let coordinates: (x: Int, y: Int, z: Int) = (3, 2, 5)

switch (coordinates) {
case (0, 0, 0): // 1
  print("Origin")
case (_, 0, 0): // 2
  print("On the x-axis.")
case (0, _, 0): // 3
  print("On the y-axis.")
case (0, 0, _): // 4
  print("On the z-axis.")
default:        // 5
  print("Somewhere in space")
}
```

1. Coincide precisamente con el caso donde el valor es (0,0,0). Este es el origen del espacio 3D.
2. Coincide con y = 0, z = 0 y cualquier valor de x. Esto significa que la coordenada está en el eje x.
3. Coincide con x = 0, z = 0 y cualquier valor de y. Esto significa que la coordenada está en ellos- eje.
4. Coincide con x = 0, y = 0 y cualquier valor de z. Esto significa que la coordenada está en el eje z.
5. Coincide con el resto de coordenadas.

Nota: usar el guión bajo para significar que no te importa el valor.

Si no quiere ignorar el valor, entonces puede usarlo en su declaración de cambio, como esto:

```
let coordinates: (x: Int, y: Int, z: Int) = (3, 2, 5)
```

```
switch (coordinates) {
case (0, 0, 0):
  print("Origin")
case (let x, 0, 0):
  print("On the x-axis at x = \(x)")
case (0, let y, 0):
  print("On the y-axis at y = \(y)")
case (0, 0, let z):
  print("On the z-axis at z = \(z)")
case (let x, let y, let z):
  print("Somewhere in space at x = \(x), y = \(y), z = \(z)")
}
```

Aquí, los casos de ejes utilizan la sintaxis de let para extraer los valores pertinentes. El código luego imprime los valores utilizando la interpolación de cadenas para construir la cadena.

Nota: no necesita un valor predeterminado en esta instrucción de cambio. Esto se debe a que el caso final es esencialmente el predeterminado: coincide con cualquier cosa, porque no hay restricciones en ninguna parte de la tupla. Si la instrucción de cambio agota todos los valores posibles con sus casos, entonces no es necesario un valor predeterminado.

También podemos usar la sintaxis de let-where para hacer coincidir casos más complejos. Por ejemplo:

```
let coordinates: (x: Int, y: Int, z: Int) = (3, 2, 5)

switch (coordinates) {
case (let x, let y, _) where y == x:
  print("Along the y = x line.")
case (let x, let y, _) where y == x * x:
  print("Along the y = x^2 line.")
default:
break
}
```

Aquí, hacemos coincidir las líneas "y es igual a x" y "y es igual a x al cuadrado".

Cambiar fracasos

Vale la pena señalar que en Swift, a diferencia de otros idiomas con los que las personas están familiarizadas, hay una ruptura implícita al final de cada declaración de caso. Para poder continuar con el siguiente caso (es decir, tener varios casos ejecutados), debe usar `fallthrough` declaración `fallthrough` .

```
switch(value) {
case 'one':
    // do operation one
    fallthrough
case 'two':
    // do this either independant, or in conjunction with first case
default:
    // default operation
}
```

Esto es útil para cosas como arroyos.

Interruptor y Enums

La instrucción Switch funciona muy bien con los valores de Enum.

```
enum CarModel {
    case Standard, Fast, VeryFast
}

let car = CarModel.Standard

switch car {
case .Standard: print("Standard")
case .Fast: print("Fast")
case .VeryFast: print("VeryFast")
}
```

> Dado que proporcionamos un caso para cada valor posible de automóvil, omitimos el caso `default` .

Interruptor y Opcionales

Algunos casos de ejemplo cuando el resultado es opcional.

```
var result: AnyObject? = someMethod()

switch result {
case nil:
    print("result is nothing")
case is String:
    print("result is a String")
case _ as Double:
    print("result is not nil, any value that is a Double")
case let myInt as Int where myInt > 0:
    print("\(myInt) value is not nil but an int and greater than 0")
case let a?:
    print("\(a) - value is unwrapped")
}
```

Interruptores y tuplas

Los interruptores pueden activar tuplas:

```
public typealias mdyTuple = (month: Int, day: Int, year: Int)

let fredsBirthday =   (month: 4, day: 3, year: 1973)




switch theMDY
{
//You can match on a literal tuple:
```

```
case (fredsBirthday):
  message = "\(date) \(prefix) the day Fred was born"

//You can match on some of the terms, and ignore others:
case (3, 15, _):
  message = "Beware the Ides of March"

//You can match on parts of a literal tuple, and copy other elements
//into a constant that you use in the body of the case:
case (bobsBirthday.month, bobsBirthday.day, let year) where year > bobsBirthday.year:
  message = "\(date) \(prefix) Bob's \(possessiveNumber(year - bobsBirthday.year))" +
    "birthday"

//You can copy one or more elements of the tuple into a constant and then
//add a where clause that further qualifies the case:
case (susansBirthday.month, susansBirthday.day, let year)
  where year > susansBirthday.year:
  message = "\(date) \(prefix) Susan's " +
    "\(possessiveNumber(year - susansBirthday.year)) birthday"

//You can match some elements to ranges:.
case (5, 1...15, let year):
  message = "\(date) \(prefix) in the first half of May, \(year)"
}
```

Coincidencia basada en clase - ideal para prepareForSegue

También puede hacer un cambio de instrucción basado en la **clase** de lo que está activando.

Un ejemplo donde esto es útil es en `prepareForSegue` . Solía cambiar según el identificador de segue, pero eso es frágil. Si cambia su guión gráfico más tarde y cambia el nombre del identificador de segue, se rompe su código. O, si usa segues para múltiples instancias en la misma clase de controlador de vista (pero diferentes escenas del guión gráfico), entonces no puede usar el identificador de segue para averiguar la clase del destino.

Cambio rápido de las declaraciones al rescate.

Use Swift `case let var as Class` sintaxis de `case let var as Class` , como esto:

3.0

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
  switch segue.destinationViewController {
    case let fooViewController as FooViewController:
      fooViewController.delegate = self

    case let barViewController as BarViewController:
      barViewController.data = data

    default:
      break
  }
}
```

3.0

En Swift 3 la sintaxis ha cambiado ligeramente:

```
override func prepare(for segue: UIStoryboardSegue, sender: AnyObject?) {
  switch segue.destinationViewController {
    case let fooViewController as FooViewController:
      fooViewController.delegate = self

    case let barViewController as BarViewController:
      barViewController.data = data

    default:
      break
  }
}
```

Lea Cambiar en línea: https://riptutorial.com/es/swift/topic/207/cambiar

Capítulo 11: Cierres

Sintaxis

- var closingVar: (<parameters>) -> (<returnType>) // Como variable o tipo de propiedad
- typealias ClosureType = (<parameters>) -> (<returnType>)
- {[<captureList>] (<parameters>) <throws-ness> -> <returnType> en <statements>} // Completa sintaxis de cierre

Observaciones

Para obtener más información sobre los cierres Swift, consulte la documentación de Apple .

Examples

Fundamentos de cierre

Los cierres (también conocidos como **bloques** o **lambdas**) son piezas de código que se pueden almacenar y transmitir dentro de su programa.

```
let sayHi = { print("Hello") }
// The type of sayHi is "() -> ()", aka "() -> Void"

sayHi()  // prints "Hello"
```

Al igual que otras funciones, los cierres pueden aceptar argumentos y devolver resultados o generar errores :

```
let addInts = { (x: Int, y: Int) -> Int in
    return x + y
}
// The type of addInts is "(Int, Int) -> Int"

let result = addInts(1, 2)  // result is 3

let divideInts = { (x: Int, y: Int) throws -> Int in
    if y == 0 {
        throw MyErrors.DivisionByZero
    }
    return x / y
}
// The type of divideInts is "(Int, Int) throws -> Int"
```

Los cierres pueden **capturar** valores de su alcance:

```
// This function returns another function which returns an integer
func makeProducer(x: Int) -> (() -> Int) {
    let closure = { x }  // x is captured by the closure
```

```
    return closure
}

// These two function calls use the exact same code,
// but each closure has captured different values.
let three = makeProducer(3)
let four = makeProducer(4)
three()  // returns 3
four()  // returns 4
```

Los cierres se pueden pasar directamente a las funciones:

```
let squares = (1...10).map({ $0 * $0 })  // returns [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
let squares = (1...10).map { $0 * $0 }

NSURLSession.sharedSession().dataTaskWithURL(myURL,
    completionHandler: { (data: NSData?, response: NSURLResponse?, error: NSError?) in
        if let data = data {
            print("Request succeeded, data: \(data)")
        } else {
            print("Request failed: \(error)")
        }
    }).resume()
```

Variaciones de sintaxis

La sintaxis básica de cierre es

> `{` `[` *lista de captura* `]` `(` *parámetros* `)` *throws-ness* `->` *return type* `in` *body* `}` .

Muchas de estas partes se pueden omitir, por lo que hay varias formas equivalentes de escribir cierres simples:

```
let addOne = { [] (x: Int) -> Int in return x + 1 }
let addOne = { [] (x: Int) -> Int in x + 1 }
let addOne = { (x: Int) -> Int in x + 1 }
let addOne = { x -> Int in x + 1 }
let addOne = { x in x + 1 }
let addOne = { $0 + 1 }

let addOneOrThrow = { [] (x: Int) throws -> Int in return x + 1 }
let addOneOrThrow = { [] (x: Int) throws -> Int in x + 1 }
let addOneOrThrow = { (x: Int) throws -> Int in x + 1 }
let addOneOrThrow = { x throws -> Int in x + 1 }
let addOneOrThrow = { x throws in x + 1 }
```

- La lista de captura se puede omitir si está vacía.
- Los parámetros no necesitan anotaciones de tipo si se pueden inferir sus tipos.
- No es necesario especificar el tipo de retorno si se puede inferir.
- Los parámetros no tienen que ser nombrados; en su lugar, se pueden referir con `$0` , `$1` , `$2` , etc.
- Si el cierre contiene una sola expresión, cuyo valor debe devolverse, se puede omitir la palabra clave `return` .
- Si se infiere que el cierre produce un error, se escribe en un contexto que espera un cierre

de lanzamiento, o si no se produce un error, se pueden omitir los `throws` .

```
// The closure's type is unknown, so we have to specify the type of x and y.
// The output type is inferred to be Int, because the + operator for Ints returns Int.
let addInts = { (x: Int, y: Int) in x + y }

// The closure's type is specified, so we can omit the parameters' type annotations.
let addInts: (Int, Int) -> Int = { x, y in x + y }
let addInts: (Int, Int) -> Int = { $0 + $1 }
```

Pasando cierres a funciones

Las funciones pueden aceptar cierres (u otras funciones) como parámetros:

```
func foo(value: Double, block: () -> Void) { ... }
func foo(value: Double, block: Int -> Int) { ... }
func foo(value: Double, block: (Int, Int) -> String) { ... }
```

Sintaxis de cierre de seguimiento

Si el último parámetro de una función es un cierre, las llaves de cierre `{` / `}` se pueden escribir **después de** la invocación de la función:

```
foo(3.5, block: { print("Hello") })

foo(3.5) { print("Hello") }

dispatch_async(dispatch_get_main_queue(), {
    print("Hello from the main queue")
})

dispatch_async(dispatch_get_main_queue()) {
    print("Hello from the main queue")
}
```

Si el único argumento de una función es un cierre, también puede omitir el par de paréntesis `()` al llamar con la sintaxis de cierre final:

```
func bar(block: () -> Void) { ... }
```

```
bar() { print("Hello") }

bar { print("Hello") }
```

Parámetros de `@noescape`

Los parámetros de cierre marcados con `@noescape` están garantizados para ejecutarse antes de que se devuelva la llamada a la función, por lo que se usa `self.` No se requiere dentro del cuerpo de cierre:

```
func executeNow(@noescape block: () -> Void) {
    // Since `block` is @noescape, it's illegal to store it to an external variable.
    // We can only call it right here.
    block()
}

func executeLater(block: () -> Void) {
    dispatch_async(dispatch_get_main_queue()) {
        // Some time in the future...
        block()
    }
}
```

```
class MyClass {
    var x = 0
    func showExamples() {
        // error: reference to property 'x' in closure requires explicit 'self.' to make
capture semantics explicit
        executeLater { x = 1 }

        executeLater { self.x = 2 }  // ok, the closure explicitly captures self

        // Here "self." is not required, because executeNow() takes a @noescape block.
        executeNow { x = 3 }

        // Again, self. is not required, because map() uses @noescape.
        [1, 2, 3].map { $0 + x }
    }
}
```

Nota de Swift 3:

Tenga en cuenta que en Swift 3, ya no marca bloques como @noescape. Los bloques ahora **no se** escapan por defecto. En Swift 3, en lugar de marcar un cierre como sin escape, se marca un parámetro de función que es un cierre con escape mediante la palabra clave "@escaping".

`throws` y `rethrows`

Los cierres, como otras funciones, pueden arrojar errores :

```
func executeNowOrIgnoreError(block: () throws -> Void) {
    do {
        try block()
    } catch {
        print("error: \(error)")
    }
}
```

La función puede, por supuesto, pasar el error a su interlocutor:

```
func executeNowOrThrow(block: () throws -> Void) throws {
    try block()
}
```

Sin embargo, si el bloque que se pasa *no se* lanza, la persona que llama sigue con la función de lanzar:

```
// It's annoying that this requires "try", because "print()" can't throw!
try executeNowOrThrow { print("Just printing, no errors here!") }
```

La solución es `rethrows` , lo que designa que la función solo se puede lanzar **si su parámetro de cierre se lanza** :

```
func executeNowOrRethrow(block: () throws -> Void) rethrows {
    try block()
}

// "try" is not required here, because the block can't throw an error.
executeNowOrRethrow { print("No errors are thrown from this closure") }

// This block can throw an error, so "try" is required.
try executeNowOrRethrow { throw MyError.Example }
```

Muchas funciones de biblioteca estándar utilizan `rethrows` , incluidos `map()` , `filter()` e `indexOf()` .

Capturas, referencias fuertes / débiles y ciclos de retención.

```
class MyClass {
    func sayHi() { print("Hello") }
    deinit { print("Goodbye") }
}
```

Cuando un cierre captura un tipo de referencia (una instancia de clase), mantiene una referencia segura por defecto:

```
let closure: () -> Void
do {
    let obj = MyClass()
    // Captures a strong reference to `obj`: the object will be kept alive
    // as long as the closure itself is alive.
    closure = { obj.sayHi() }
    closure()  // The object is still alive; prints "Hello"
} // obj goes out of scope
closure()  // The object is still alive; prints "Hello"
```

La **lista de captura** del cierre se puede usar para especificar una referencia débil o sin dueño:

```
let closure: () -> Void
do {
    let obj = MyClass()
    // Captures a weak reference to `obj`: the closure will not keep the object alive;
    // the object becomes optional inside the closure.
    closure = { [weak obj] in obj?.sayHi() }
    closure()  // The object is still alive; prints "Hello"
} // obj goes out of scope and is deallocated; prints "Goodbye"
closure()  // `obj` is nil from inside the closure; this does not print anything.
```

```
let closure: () -> Void
do {
    let obj = MyClass()
    // Captures an unowned reference to `obj`: the closure will not keep the object alive;
    // the object is always assumed to be accessible while the closure is alive.
    closure = { [unowned obj] in obj.sayHi() }
    closure()  // The object is still alive; prints "Hello"
} // obj goes out of scope and is deallocated; prints "Goodbye"
closure()  // crash! obj is being accessed after it's deallocated.
```

Para obtener más información, consulte el tema Administración de memoria y la sección Recuento automático de referencias de The Swift Programming Language.

Retener ciclos

Si un objeto mantiene un cierre, que también contiene una fuerte referencia al objeto, este es un **ciclo de retención** . A menos que el ciclo se rompa, la memoria que almacena el objeto y el cierre se filtrará (nunca se recuperará).

```
class Game {
    var score = 0
    let controller: GCController
    init(controller: GCController) {
        self.controller = controller

        // BAD: the block captures self strongly, but self holds the controller
        // (and thus the block) strongly, which is a cycle.
        self.controller.controllerPausedHandler = {
            let curScore = self.score
            print("Pause button pressed; current score: \(curScore)")
        }

        // SOLUTION: use `weak self` to break the cycle.
        self.controller.controllerPausedHandler = { [weak self] in
            guard let strongSelf = self else { return }
            let curScore = strongSelf.score
            print("Pause button pressed; current score: \(curScore)")
        }
    }
}
```

Utilización de cierres para codificación asíncrona.

Los cierres a menudo se utilizan para tareas asíncronas, por ejemplo, cuando se obtienen datos de un sitio web.

3.0

```
func getData(urlString: String, callback: (result: NSData?) -> Void) {

    // Turn the URL string into an NSURLRequest.
    guard let url = NSURL(string: urlString) else { return }
    let request = NSURLRequest(URL: url)
```

```
    // Asynchronously fetch data from the given URL.
    let task = NSURLSession.sharedSession().dataTaskWithRequest(request) {(data: NSData?,
response: NSURLResponse?, error: NSError?) in

        // We now have the NSData response from the website.
        // We can get it "out" of the function by using the callback
        // that was passed to this function as a parameter.

        callback(result: data)
    }

    task.resume()
}
```

Esta función es asíncrona, por lo que no bloqueará el subproceso al que se está llamando (no se congelará la interfaz si se llama en el subproceso principal de su aplicación GUI).

3.0

```
print("1. Going to call getData")

getData("http://www.example.com") {(result: NSData?) -> Void in

    // Called when the data from http://www.example.com has been fetched.
    print("2. Fetched data")
}

print("3. Called getData")
```

Debido a que la tarea es asíncrona, la salida generalmente se verá así:

```
"1. Going to call getData"
"3. Called getData"
"2. Fetched data"
```

Debido a que el código dentro del cierre, `print("2. Fetched data")` , no se llamará hasta que se `print("2. Fetched data")` los datos de la URL.

Cierres y alias de tipo

Un cierre puede definirse con un `typealias` . Esto proporciona un tipo de marcador de posición conveniente si se utiliza la misma firma de cierre en varios lugares. Por ejemplo, las devoluciones de llamadas de solicitud de red comunes o los controladores de eventos de interfaz de usuario son excelentes candidatos para ser "nombrados" con un alias de tipo.

```
public typealias ClosureType = (x: Int, y: Int) -> Int
```

A continuación, puede definir una función utilizando las tipografías:

```
public func closureFunction(closure: ClosureType) {
    let z = closure(1, 2)
}
```

```
closureFunction() { (x: Int, y: Int) -> Int in return x + y }
```

Lea Cierres en línea: https://riptutorial.com/es/swift/topic/262/cierres

Capítulo 12: Cifrado AES

Examples

Cifrado AES en modo CBC con un IV aleatorio (Swift 3.0)

El iv está prefijado a los datos encriptados.

`aesCBC128Encrypt` creará un IV aleatorio y con prefijo al código encriptado.
`aesCBC128Decrypt` usará el prefijo IV durante el descifrado.

Las entradas son los datos y la clave son los objetos de datos. Si una forma codificada, como Base64, si es necesario, se convierte y / o en el método de llamada.

La clave debe tener exactamente 128 bits (16 bytes), 192 bits (24 bytes) o 256 bits (32 bytes) de longitud. Si se usa otro tamaño de clave, se lanzará un error.

El relleno PKCS # 7 se establece de forma predeterminada.

Este ejemplo requiere Crypto común
Es necesario tener un encabezado puente al proyecto:
`#import <CommonCrypto/CommonCrypto.h>`
Agregue el `Security.framework` al proyecto.

Este es un ejemplo, no un código de producción.

```
enum AESError: Error {
    case KeyError((String, Int))
    case IVError((String, Int))
    case CryptorError((String, Int))
}

// The iv is prefixed to the encrypted data
func aesCBCEncrypt(data:Data, keyData:Data) throws -> Data {
    let keyLength = keyData.count
    let validKeyLengths = [kCCKeySizeAES128, kCCKeySizeAES192, kCCKeySizeAES256]
    if (validKeyLengths.contains(keyLength) == false) {
        throw AESError.KeyError(("Invalid key length", keyLength))
    }

    let ivSize = kCCBlockSizeAES128;
    let cryptLength = size_t(ivSize + data.count + kCCBlockSizeAES128)
    var cryptData = Data(count:cryptLength)

    let status = cryptData.withUnsafeMutableBytes {ivBytes in
        SecRandomCopyBytes(kSecRandomDefault, kCCBlockSizeAES128, ivBytes)
    }
    if (status != 0) {
        throw AESError.IVError(("IV generation failed", Int(status)))
    }

    var numBytesEncrypted :size_t = 0
```

```
    let options   = CCOptions(kCCOptionPKCS7Padding)

    let cryptStatus = cryptData.withUnsafeMutableBytes {cryptBytes in
        data.withUnsafeBytes {dataBytes in
            keyData.withUnsafeBytes {keyBytes in
                CCCrypt(CCOperation(kCCEncrypt),
                        CCAlgorithm(kCCAlgorithmAES),
                        options,
                        keyBytes, keyLength,
                        cryptBytes,
                        dataBytes, data.count,
                        cryptBytes+kCCBlockSizeAES128, cryptLength,
                        &numBytesEncrypted)
            }
        }
    }

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        cryptData.count = numBytesEncrypted + ivSize
    }
    else {
        throw AESError.CryptorError(("Encryption failed", Int(cryptStatus)))
    }

    return cryptData;
}

// The iv is prefixed to the encrypted data
func aesCBCDecrypt(data:Data, keyData:Data) throws -> Data? {
    let keyLength = keyData.count
    let validKeyLengths = [kCCKeySizeAES128, kCCKeySizeAES192, kCCKeySizeAES256]
    if (validKeyLengths.contains(keyLength) == false) {
        throw AESError.KeyError(("Invalid key length", keyLength))
    }

    let ivSize = kCCBlockSizeAES128;
    let clearLength = size_t(data.count - ivSize)
    var clearData = Data(count:clearLength)

    var numBytesDecrypted :size_t = 0
    let options   = CCOptions(kCCOptionPKCS7Padding)

    let cryptStatus = clearData.withUnsafeMutableBytes {cryptBytes in
        data.withUnsafeBytes {dataBytes in
            keyData.withUnsafeBytes {keyBytes in
                CCCrypt(CCOperation(kCCDecrypt),
                        CCAlgorithm(kCCAlgorithmAES128),
                        options,
                        keyBytes, keyLength,
                        dataBytes,
                        dataBytes+kCCBlockSizeAES128, clearLength,
                        cryptBytes, clearLength,
                        &numBytesDecrypted)
            }
        }
    }

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        clearData.count = numBytesDecrypted
    }
    else {
```

```
        throw AESError.CryptorError(("Decryption failed", Int(cryptStatus)))
    }

    return clearData;
}
```

Ejemplo de uso:

```
let clearData = "clearData0123456".data(using:String.Encoding.utf8)!
let keyData   = "keyData890123456".data(using:String.Encoding.utf8)!
print("clearData:   \(clearData as NSData)")
print("keyData:     \(keyData as NSData)")

var cryptData :Data?
do {
    cryptData = try aesCBCEncrypt(data:clearData, keyData:keyData)
    print("cryptData:   \(cryptData! as NSData)")
}
catch (let status) {
    print("Error aesCBCEncrypt: \(status)")
}

let decryptData :Data?
do {
    let decryptData = try aesCBCDecrypt(data:cryptData!, keyData:keyData)
    print("decryptData: \(decryptData! as NSData)")
}
catch (let status) {
    print("Error aesCBCDecrypt: \(status)")
}
```

Ejemplo de salida:

```
clearData:   <636c6561 72446174 61303132 33343536>
keyData:     <6b657944 61746138 39303132 33343536>
cryptData:   <92c57393 f454d959 5a4d158f 6e1cd3e7 77986ee9 b2970f49 2bafcf1a 8ee9d51a bde49c31
d7780256 71837a61 60fa4be0>
decryptData: <636c6561 72446174 61303132 33343536>
```

Notas:
Un problema típico con el código de ejemplo del modo CBC es que deja la creación y el intercambio del IV aleatorio al usuario. Este ejemplo incluye la generación del IV, prefijó los datos cifrados y usa el IV prefijado durante el descifrado. Esto libera al usuario ocasional de los detalles que son necesarios para el modo CBC .

Por seguridad, los datos cifrados también deben tener autenticación, este código de ejemplo no proporciona eso para ser pequeño y permitir una mejor interoperabilidad para otras plataformas.

También falta la derivación clave de la clave de una contraseña, se sugiere que se use PBKDF2 si se usan contraseñas de texto como material de claves.

Para un robusto código de encriptación multiplataforma listo para producción, consulte RNCryptor

Actualizado para usar tirar / atrapar y múltiples tamaños de clave basados en la clave

provista.

Cifrado AES en modo CBC con un IV aleatorio (Swift 2.3)

El iv está prefijado a los datos encriptados.

aesCBC128Encrypt creará un IV aleatorio y con prefijo al código encriptado. aesCBC128Decrypt usará el prefijo IV durante el descifrado.

Las entradas son los datos y la clave son los objetos de datos. Si una forma codificada, como Base64, si es necesario, se convierte y / o en el método de llamada.

La clave debe ser exactamente de 128 bits (16 bytes). Para otros tamaños de clave vea el ejemplo de Swift 3.0.

El relleno PKCS # 7 se establece de forma predeterminada.

Este ejemplo requiere Common Crypto. Es necesario tener un encabezado puente para el proyecto: #import <CommonCrypto / CommonCrypto.h> Agregue el archivo Security.framework al proyecto.

Vea el ejemplo de Swift 3 para notas.

Este es un ejemplo, no un código de producción.

```
func aesCBC128Encrypt(data data:[UInt8], keyData:[UInt8]) -> [UInt8]? {
    let keyLength   = size_t(kCCKeySizeAES128)
    let ivLength    = size_t(kCCBlockSizeAES128)
    let cryptDataLength = size_t(data.count + kCCBlockSizeAES128)
    var cryptData = [UInt8](count:ivLength + cryptDataLength, repeatedValue:0)

    let status = SecRandomCopyBytes(kSecRandomDefault, Int(ivLength),
UnsafeMutablePointer<UInt8>(cryptData));
    if (status != 0) {
        print("IV Error, errno: \(status)")
        return nil
    }

    var numBytesEncrypted :size_t = 0
    let cryptStatus = CCCrypt(CCOperation(kCCEncrypt),
                              CCAlgorithm(kCCAlgorithmAES128),
                              CCOptions(kCCOptionPKCS7Padding),
                              keyData, keyLength,
                              cryptData,
                              data, data.count,
                              &cryptData + ivLength, cryptDataLength,
                              &numBytesEncrypted)

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        cryptData.removeRange(numBytesEncrypted+ivLength..<cryptData.count)
    }
    else {
        print("Error: \(cryptStatus)")
        return nil;
    }
```

```
    return cryptData;
}

func aesCBC128Decrypt(data data:[UInt8], keyData:[UInt8]) -> [UInt8]? {
    let clearLength = size_t(data.count)
    var clearData   = [UInt8](count:clearLength, repeatedValue:0)

    let keyLength   = size_t(kCCKeySizeAES128)
    let ivLength    = size_t(kCCBlockSizeAES128)

    var numBytesDecrypted :size_t = 0
    let cryptStatus = CCCrypt(CCOperation(kCCDecrypt),
                              CCAlgorithm(kCCAlgorithmAES128),
                              CCOptions(kCCOptionPKCS7Padding),
                              keyData, keyLength,
                              data,
                              UnsafePointer<UInt8>(data) + ivLength, data.count - ivLength,
                              &clearData, clearLength,
                              &numBytesDecrypted)

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        clearData.removeRange(numBytesDecrypted..<clearLength)

    } else {
        print("Error: \(cryptStatus)")
        return nil;
    }

    return clearData;
}
```

Ejemplo de uso:

```
let clearData = toData("clearData0123456")
let keyData   = toData("keyData890123456")

print("clearData:   \(toHex(clearData))")
print("keyData:     \(toHex(keyData))")
let cryptData = aesCBC128Encrypt(data:clearData, keyData:keyData)!
print("cryptData:   \(toHex(cryptData))")
let decryptData = aesCBC128Decrypt(data:cryptData, keyData:keyData)!
print("decryptData: \(toHex(decryptData))")
```

Ejemplo de salida:

```
clearData:   <636c6561 72446174 61303132 33343536>
keyData:     <6b657944 61746138 39303132 33343536>
cryptData:   <9fce4323 830e3734 93dd93bf e464f72a a653a3a5 2c40d5ea e90c1017 958750a7 ff094c53
6a81b458 b1fbd6d4 1f583298>
decryptData: <636c6561 72446174 61303132 33343536>
```

Cifrado AES en modo ECB con relleno PKCS7

De la documentación de Apple para IV,

> Este parámetro se ignora si se usa el modo ECB o si se selecciona un algoritmo de

cifrado de flujo.

```
func AESEncryption(key: String) -> String? {

        let keyData: NSData! = (key as NSString).data(using: String.Encoding.utf8.rawValue) as
NSData!

        let data: NSData! = (self as NSString).data(using: String.Encoding.utf8.rawValue) as
NSData!

        let cryptData    = NSMutableData(length: Int(data.length) + kCCBlockSizeAES128)!

        let keyLength              = size_t(kCCKeySizeAES128)
        let operation: CCOperation = UInt32(kCCEncrypt)
        let algoritm:  CCAlgorithm = UInt32(kCCAlgorithmAES128)
        let options:   CCOptions   = UInt32(kCCOptionECBMode + kCCOptionPKCS7Padding)

        var numBytesEncrypted :size_t = 0


        let cryptStatus = CCCrypt(operation,
                                  algoritm,
                                  options,
                                  keyData.bytes, keyLength,
                                  nil,
                                  data.bytes, data.length,
                                  cryptData.mutableBytes, cryptData.length,
                                  &numBytesEncrypted)

        if UInt32(cryptStatus) == UInt32(kCCSuccess) {
            cryptData.length = Int(numBytesEncrypted)

            var bytes = [UInt8](repeating: 0, count: cryptData.length)
            cryptData.getBytes(&bytes, length: cryptData.length)

            var hexString = ""
            for byte in bytes {
                hexString += String(format:"%02x", UInt8(byte))
            }

            return hexString
        }

        return nil
    }
```

Lea Cifrado AES en línea: https://riptutorial.com/es/swift/topic/7026/cifrado-aes

Capítulo 13: Comenzando con la Programación Orientada al Protocolo

Observaciones

Para obtener más información sobre este tema, consulte la Programación orientada a protocolos de WWDC 2015 talk en Swift .

También hay una gran guía escrita sobre el mismo: Introducción a la programación orientada a protocolos en Swift 2 .

Examples

Aprovechamiento de la programación orientada al protocolo para pruebas unitarias

La programación orientada al protocolo es una herramienta útil para escribir fácilmente mejores pruebas unitarias para nuestro código.

Digamos que queremos probar un UIViewController que se basa en una clase ViewModel.

Los pasos necesarios en el código de producción son:

1. Defina un protocolo que exponga la interfaz pública de la clase ViewModel, con todas las propiedades y métodos que necesita el UIViewController.
2. Implementar la clase de ViewModel real, de acuerdo con ese protocolo.
3. Use una técnica de inyección de dependencia para permitir que el controlador de vista use la implementación que queremos, pasándola como protocolo y no como instancia concreta.

```
protocol ViewModelType {
   var title : String {get}
   func confirm()
}

class ViewModel : ViewModelType {
   let title : String

   init(title: String) {
       self.title = title
   }
   func confirm() { ... }
}

class ViewController : UIViewController {
   // We declare the viewModel property as an object conforming to the protocol
   // so we can swap the implementations without any friction.
   var viewModel : ViewModelType!
   @IBOutlet var titleLabel : UILabel!
```

```
    override func viewDidLoad() {
        super.viewDidLoad()
        titleLabel.text = viewModel.title
    }

    @IBAction func didTapOnButton(sender: UIButton) {
        viewModel.confirm()
    }
}

// With DI we setup the view controller and assign the view model.
// The view controller doesn't know the concrete class of the view model,
// but just relies on the declared interface on the protocol.
let viewController = //... Instantiate view controller
viewController.viewModel = ViewModel(title: "MyTitle")
```

Luego, en la prueba de la unidad:

1. Implementar un modelo de vista simulado que se ajuste al mismo protocolo.
2. Páselo al UIViewController en prueba utilizando la inyección de dependencia, en lugar de la instancia real.
3. ¡Prueba!

```
class FakeViewModel : ViewModelType {
    let title : String = "FakeTitle"

    var didConfirm = false
    func confirm() {
        didConfirm = true
    }
}

class ViewControllerTest : XCTestCase {
     var sut : ViewController!
     var viewModel : FakeViewModel!

     override func setUp() {
         super.setUp()

         viewModel = FakeViewModel()
         sut = // ... initialization for view controller
         sut.viewModel = viewModel

         XCTAssertNotNil(self.sut.view) // Needed to trigger view loading
     }

     func testTitleLabel() {
         XCTAssertEqual(self.sut.titleLabel.text, "FakeTitle")
     }

     func testTapOnButton() {
         sut.didTapOnButton(UIButton())
         XCTAssertTrue(self.viewModel.didConfirm)
     }
}
```

Usando protocolos como tipos de primera clase.

La programación orientada a protocolos se puede utilizar como un patrón de diseño Swift central.

Los diferentes tipos pueden ajustarse al mismo protocolo, los tipos de valor pueden incluso adaptarse a múltiples protocolos e incluso proporcionar la implementación del método predeterminado.

Inicialmente, se definen protocolos que pueden representar propiedades y / o métodos de uso común con tipos específicos o genéricos.

```
protocol ItemData {

    var title: String { get }
    var description: String { get }
    var thumbnailURL: NSURL { get }
    var created: NSDate { get }
    var updated: NSDate { get }

}

protocol DisplayItem {

    func hasBeenUpdated() -> Bool
    func getFormattedTitle() -> String
    func getFormattedDescription() -> String

}

protocol GetAPIItemDataOperation {

    static func get(url: NSURL, completed: ([ItemData]) -> Void)
}
```

Se puede crear una implementación predeterminada para el método get, aunque si se desea, los tipos conformes pueden anular la implementación.

```
extension GetAPIItemDataOperation {

    static func get(url: NSURL, completed: ([ItemData]) -> Void) {

        let date = NSDate(
        timeIntervalSinceNow: NSDate().timeIntervalSince1970
            + 5000)

        // get data from url
        let urlData: [String: AnyObject] = [
            "title": "Red Camaro",
            "desc": "A fast red car.",
            "thumb":"http://cars.images.com/red-camaro.png",
            "created": NSDate(), "updated": date]

        // in this example forced unwrapping is used
        // forced unwrapping should never be used in practice
        // instead conditional unwrapping should be used (guard or if/let)
        let item = Item(
            title: urlData["title"] as! String,
            description: urlData["desc"] as! String,
            thumbnailURL: NSURL(string: urlData["thumb"] as! String)!,
```

```
            created: urlData["created"] as! NSDate,
            updated: urlData["updated"] as! NSDate)

        completed([item])

    }
}

struct ItemOperation: GetAPIItemDataOperation { }
```

Un tipo de valor que se ajusta al protocolo ItemData, este tipo de valor también puede ajustarse a otros protocolos.

```
struct Item: ItemData {

    let title: String
    let description: String
    let thumbnailURL: NSURL
    let created: NSDate
    let updated: NSDate

}
```

Aquí la estructura del elemento se amplía para ajustarse a un elemento de visualización.

```
extension Item: DisplayItem {

    func hasBeenUpdated() -> Bool {
        return updated.timeIntervalSince1970 >
            created.timeIntervalSince1970
    }

    func getFormattedTitle() -> String {
        return title.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }

    func getFormattedDescription() -> String {
        return description.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }
}
```

Un sitio de llamada de ejemplo para usar el método estático de obtención.

```
ItemOperation.get(NSURL()) { (itemData) in

    // perhaps inform a view of new data
    // or parse the data for user requested info, etc.
    dispatch_async(dispatch_get_main_queue(), {

        // self.items = itemData
    })

}
```

Diferentes casos de uso requerirán diferentes implementaciones. La idea principal aquí es mostrar la conformidad de diversos tipos en los que el protocolo es el punto principal del enfoque en el diseño. En este ejemplo, tal vez los datos de la API se guardan condicionalmente en una entidad de Datos Core.

```
// the default core data created classes + extension
class LocalItem: NSManagedObject { }

extension LocalItem {

    @NSManaged var title: String
    @NSManaged var itemDescription: String
    @NSManaged var thumbnailURLStr: String
    @NSManaged var createdAt: NSDate
    @NSManaged var updatedAt: NSDate
}
```

Aquí, la clase respaldada por Core Data también puede ajustarse al protocolo DisplayItem.

```
extension LocalItem: DisplayItem {

    func hasBeenUpdated() -> Bool {
        return updatedAt.timeIntervalSince1970 >
            createdAt.timeIntervalSince1970
    }

    func getFormattedTitle() -> String {
        return title.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }

    func getFormattedDescription() -> String {
        return itemDescription.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }
}

// In use, the core data results can be
// conditionally casts as a protocol
class MyController: UIViewController {

    override func viewDidLoad() {

        let fr: NSFetchRequest = NSFetchRequest(
        entityName: "Items")

        let context = NSManagedObjectContext(
        concurrencyType: .MainQueueConcurrencyType)

        do {

            let items: AnyObject = try context.executeFetchRequest(fr)
            if let displayItems = items as? [DisplayItem] {

                print(displayItems)
            }

        } catch let error as NSError {
```

```
            print(error.localizedDescription)
        }

    }
}
```

Lea Comenzando con la Programación Orientada al Protocolo en línea:
https://riptutorial.com/es/swift/topic/2502/comenzando-con-la-programacion-orientada-al-protocolo

Capítulo 14: Concurrencia

Sintaxis

- Swift 3.0
- DispatchQueue.main // Obtener la cola principal
- DispatchQueue (etiqueta: "my-serial-queue", atributos: [.serial, .qosBackground]) // Crea tu propia cola serial privada
- DispatchQueue.global (atributos: [.qosDefault]) // Accede a una de las colas simultáneas globales
- DispatchQueue.main.async {...} // Distribuye una tarea de forma asíncrona al hilo principal
- DispatchQueue.main.sync {...} // Distribuye una tarea de forma sincrónica al hilo principal
- DispatchQueue.main.asyncAfter (fecha límite: .now () + 3) {...} // Distribuye una tarea de forma asíncrona al hilo principal que se ejecutará después de x segundos
- Swift <3.0
- dispatch_get_main_queue () // Obtenga la cola principal ejecutándose en el hilo principal
- dispatch_get_global_queue (dispatch_queue_priority_t, 0) // Obtener cola global con prioridad especificada dispatch_queue_priority_t
- dispatch_async (dispatch_queue_t) {() -> Void in ...} // Distribuye una tarea de forma asíncrona en la dispatch_queue_t especificada
- dispatch_sync (dispatch_queue_t) {() -> Void in ...} // Distribuye una tarea de forma síncrona en el dispatch_queue_t especificado
- dispatch_after (dispatch_time (DISPATCH_TIME_NOW, Int64 (nanosegundos)), dispatch_queue_t, {...}); // Enviar una tarea en la dispatch_queue_t especificada después de nanosegundos

Examples

Obtención de una cola de Grand Central Dispatch (GCD)

Grand Central Dispatch trabaja en el concepto de "Dispatch Queues". Una cola de envío ejecuta las tareas que usted designa en el orden en que se pasan. Hay tres tipos de colas de despacho:

- **Serial Dispatch Queues** (colas de envío privado) se ejecutan una tarea a la vez, en orden. Se utilizan con frecuencia para sincronizar el acceso a un recurso.

- **Las colas de despacho simultáneas** (también conocidas como colas de despacho globales) ejecutan una o más tareas simultáneamente.
- La **cola de envío principal** ejecuta tareas en el hilo principal.

Para acceder a la cola principal:

3.0

```
let mainQueue = DispatchQueue.main
```

3.0

```
let mainQueue = dispatch_get_main_queue()
```

El sistema proporciona colas de despacho globales *simultáneas* (globales a su aplicación), con diferentes prioridades. Puede acceder a estas colas utilizando la clase `DispatchQueue` en Swift 3:

3.0

```
let globalConcurrentQueue = DispatchQueue.global(qos: .default)
```

equivalente a

```
let globalConcurrentQueue = DispatchQueue.global()
```

3.0

```
let globalConcurrentQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)
```

En iOS 8 o posterior, los posibles valores de calidad de servicio que se pueden pasar son `.userInteractive` , `.userInitiated` , `.default` , `.utility` y `.background` . Estos reemplazan las constantes `DISPATCH_QUEUE_PRIORITY_` .

También puede crear sus propias colas con diferentes prioridades:

3.0

```
let myConcurrentQueue = DispatchQueue(label: "my-concurrent-queue", qos: .userInitiated,
attributes: [.concurrent], autoreleaseFrequency: .workItem, target: nil)
let mySerialQueue = DispatchQueue(label: "my-serial-queue", qos: .background, attributes: [],
autoreleaseFrequency: .workItem, target: nil)
```

3.0

```
let myConcurrentQueue = dispatch_queue_create("my-concurrent-queue",
DISPATCH_QUEUE_CONCURRENT)
let mySerialQueue = dispatch_queue_create("my-serial-queue", DISPATCH_QUEUE_SERIAL)
```

En Swift 3, las colas creadas con este inicializador son en serie de forma predeterminada, y pasar `.workItem` para la frecuencia de liberación automática garantiza que se crea y se vacía un grupo

de autorelease para cada elemento de trabajo. También existe `.never` , lo que significa que usted mismo administrará sus propios grupos de autorelease, o `.inherit` que hereda la configuración del entorno. En la mayoría de los casos, es probable que no use `.never` excepto en casos de personalización extrema.

Ejecución de tareas en una cola de Grand Central Dispatch (GCD)

3.0

Para ejecutar tareas en una cola de envío, use los métodos `sync` , `async` y `after` .

Para enviar una tarea a una cola de forma asíncrona:

```
let queue = DispatchQueue(label: "myQueueName")

queue.async {
    //do something

    DispatchQueue.main.async {
        //this will be called in main thread
        //any UI updates should be placed here
    }
}
// ... code here will execute immediately, before the task finished
```

Para enviar una tarea a una cola de forma síncrona:

```
queue.sync {
    // Do some task
}
// ... code here will not execute until the task is finished
```

Para enviar una tarea a una cola después de un cierto número de segundos:

```
queue.asyncAfter(deadline: .now() + 3) {
    //this will be executed in a background-thread after 3 seconds
}
// ... code here will execute immediately, before the task finished
```

> **NOTA:** ¡ Cualquier actualización de la interfaz de usuario debe llamarse en el hilo principal! Asegúrese de que coloca el código para las actualizaciones de la interfaz de usuario dentro de `DispatchQueue.main.async { ... }`

2.0

Tipos de cola:

```
let mainQueue = dispatch_get_main_queue()
let highQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0)
let backgroundQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0)
```

Para enviar una tarea a una cola de forma asíncrona:

```
dispatch_async(queue) {
    // Your code run run asynchronously. Code is queued and executed
    // at some point in the future.
}
// Code after the async block will execute immediately
```

Para enviar una tarea a una cola de forma síncrona:

```
dispatch_sync(queue) {
    // Your sync code
}
// Code after the sync block will wait until the sync task finished
```

Para enviar una tarea después de un intervalo de tiempo (use `NSEC_PER_SEC` para convertir segundos a nanosegundos):

```
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, Int64(2.5 * Double(NSEC_PER_SEC))),
dispatch_get_main_queue()) {
    // Code to be performed in 2.5 seconds here
}
```

Para ejecutar una tarea de forma asíncrona y luego actualizar la interfaz de usuario:

```
dispatch_async(queue) {
    // Your time consuming code here
    dispatch_async(dispatch_get_main_queue()) {
        // Update the UI code
    }
}
```

> **NOTA:** ¡ Cualquier actualización de la interfaz de usuario debe llamarse en el hilo principal! Asegúrese de que coloca el código para las actualizaciones de la interfaz de usuario dentro de `dispatch_async(dispatch_get_main_queue()) { ... }`

Loops concurrentes

GCD proporciona un mecanismo para realizar un bucle, por lo que los bucles se producen simultáneamente entre sí. Esto es muy útil cuando se realizan una serie de cálculos computacionalmente costosos.

Considere este bucle:

```
for index in 0 ..< iterations {
    // Do something computationally expensive here
}
```

Puede realizar esos cálculos al mismo tiempo utilizando `concurrentPerform` (en Swift 3) o `dispatch_apply` (en Swift 2):

3.0

```
DispatchQueue.concurrentPerform(iterations: iterations) { index in
    // Do something computationally expensive here
}
```

3.0

```
dispatch_apply(iterations, queue) { index in
    // Do something computationally expensive here
}
```

El cierre del bucle se invocará para cada `index` de `0` a, pero sin incluir, `iterations` . Estas iteraciones se ejecutarán simultáneamente entre sí, y por lo tanto el orden en que se ejecutan no está garantizado. El número real de iteraciones que ocurren simultáneamente en un momento dado generalmente viene determinado por las capacidades del dispositivo en cuestión (por ejemplo, cuántos núcleos tiene el dispositivo).

Un par de consideraciones especiales:

- `concurrentPerform` / `dispatch_apply` puede ejecutar los bucles simultáneamente entre sí, pero todo esto sucede de forma sincrónica con respecto al subproceso desde el que lo llama. Por lo tanto, no lo llames desde el hilo principal, ya que bloqueará ese hilo hasta que se complete el ciclo.

- Debido a que estos bucles se producen simultáneamente entre sí, usted es responsable de garantizar la seguridad de los resultados. Por ejemplo, si actualiza algún diccionario con los resultados de estos cálculos computacionalmente costosos, asegúrese de sincronizar esas actualizaciones usted mismo.

- Tenga en cuenta que hay algunos gastos generales asociados con la ejecución de bucles simultáneos. Por lo tanto, si los cálculos que se realizan dentro del bucle no son lo suficientemente intensivos en computación, puede encontrar que cualquier rendimiento obtenido mediante el uso de bucles concurrentes puede disminuir, si no compensarse completamente, por la sobrecarga asociada con la sincronización de todos estos subprocesos concurrentes.

  Por lo tanto, usted es responsable de determinar la cantidad correcta de trabajo a realizar en cada iteración del bucle. Si los cálculos son demasiado simples, puede emplear "zancada" para incluir más trabajo por bucle. Por ejemplo, en lugar de hacer un bucle concurrente con 1 millón de cálculos triviales, puede hacer 100 iteraciones en su bucle, haciendo 10.000 cálculos por bucle. De esa manera, se realiza suficiente trabajo en cada subproceso, por lo que la sobrecarga asociada con la administración de estos bucles simultáneos se vuelve menos significativa.

Ejecutar tareas en un OperationQueue

Puede pensar en un `OperationQueue` como una línea de tareas que esperan ser ejecutadas. A diferencia de las colas de despacho en GCD, las colas de operación no son FIFO (primero en entrar, primero en salir). En su lugar, ejecutan tareas tan pronto como están listas para

ejecutarse, siempre que haya suficientes recursos del sistema para permitirlo.

Obtener el principal `OperationQueue` :

3.0

```
let mainQueue = OperationQueue.main
```

Crear un `OperationQueue` personalizado:

3.0

```
let queue = OperationQueue()
queue.name = "My Queue"
queue.qualityOfService = .default
```

La calidad del servicio especifica la importancia del trabajo o la cantidad de probabilidades de que el usuario cuente con los resultados inmediatos de la tarea.

Agregar una `Operation` a un `OperationQueue` :

3.0

```
// An instance of some Operation subclass
let operation = BlockOperation {
    // perform task here
}

queue.addOperation(operation)
```

Añadir un bloque a un `OperationQueue` :

3.0

```
myQueue.addOperation {
    // some task
}
```

Agregue múltiples `Operation` a un `OperationQueue` :

3.0

```
let operations = [Operation]()
// Fill array with Operations

myQueue.addOperation(operations)
```

Ajuste la cantidad de `Operation` se pueden ejecutar simultáneamente en la cola:

```
myQueue.maxConcurrentOperationCount = 3 // 3 operations may execute at once

// Sets number of concurrent operations based on current system conditions
myQueue.maxConcurrentOperationCount = NSOperationQueueDefaultMaxConcurrentOperationCount
```

La suspensión de una cola evitará que comience la ejecución de cualquier operación existente, no iniciada o de cualquier nueva operación agregada a la cola. La forma de reanudar esta cola es establecer `isSuspended` nuevo en `false` :

3.0

```
myQueue.isSuspended = true

// Re-enable execution
myQueue.isSuspended = false
```

La suspensión de un `OperationQueue` no detiene o cancela las operaciones que ya se están ejecutando. Uno solo debe intentar suspender una cola que usted creó, no las colas globales o la cola principal.

Creación de operaciones de alto nivel

Foundation Framework proporciona el tipo de `Operation` , que representa un objeto de alto nivel que encapsula una parte del trabajo que puede ejecutarse en una cola. La cola no solo coordina el rendimiento de esas operaciones, sino que también puede establecer dependencias entre operaciones, crear operaciones cancelables, restringir el grado de concurrencia empleado por la cola de operaciones, etc.

`Operation` están listas para ejecutarse cuando todas sus dependencias terminan de ejecutarse. La propiedad `isReady` cambia a `true` .

Cree una subclase de `Operation` no concurrente simple:

3.0

```
class MyOperation: Operation {

    init(<parameters>) {
        // Do any setup work here
    }

    override func main() {
        // Perform the task
    }

}
```

2.3

```
class MyOperation: NSOperation {

    init(<parameters>) {
        // Do any setup work here
    }

    override func main() {
        // Perform the task
    }
```

```
}
```

Agregue una operación a un `OperationQueue` :

1.0

```
myQueue.addOperation(operation)
```

Esto ejecutará la operación simultáneamente en la cola.

Gestionar dependencias en una `Operation` .

Las dependencias definen otras `Operation` que deben ejecutarse en una cola antes de que esa `Operation` se considere lista para ejecutarse.

1.0

```
operation2.addDependency(operation1)

operation2.removeDependency(operation1)
```

Ejecutar una `Operation` sin una cola:

1.0

```
   operation.start()
```

Las dependencias serán ignoradas. Si se trata de una operación simultánea, la tarea aún puede ejecutarse simultáneamente si las descargas de su método de `start` funcionan en colas en segundo plano.

Operaciones concurrentes.

Si la tarea que debe realizar una `Operation` es, en sí misma, asíncrona, (por ejemplo, una tarea de datos `URLSession`), debe implementar la `Operation` como una operación concurrente. En este caso, su implementación `isAsynchronous` debería devolver `true` , generalmente tendría un método de `start` que realiza alguna configuración y luego llama a su método `main` que ejecuta la tarea.

Cuando se inicia la implementación de una `Operation` asíncrona, debe implementar `isExecuting` , `isFinished` y KVO. Entonces, cuando se inicia la ejecución, `isExecuting` propiedad cambia a `true` . Cuando una `Operation` finaliza su tarea, `isExecuting` se establece en `false` y `isFinished` se establece en `true` . Si la operación se cancela, se cancela tanto el `isCancelled` como el cambio `isFinished` en `true` . Todas estas propiedades son clave-valor observable.

Cancelar una `Operation` .

Llamar `cancel` simplemente cambia la propiedad `isCancelled` a `true` . Para responder a la cancelación desde su propia subclase de `Operation` , debe verificar el valor de `isCancelled` al menos periódicamente dentro de `main` y responder adecuadamente.

1.0

```
operation.cancel()
```

Lea Concurrencia en línea: https://riptutorial.com/es/swift/topic/1649/concurrencia

Capítulo 15: Condicionales

Introducción

Las expresiones condicionales, que incluyen palabras clave como if, else if, y else, brindan a los programas Swift la capacidad de realizar diferentes acciones dependiendo de una condición booleana: Verdadero o Falso. Esta sección cubre el uso de condicionales Swift, lógica booleana y declaraciones ternarias.

Observaciones

Para obtener más información sobre sentencias condicionales, consulte El lenguaje de programación Swift .

Examples

Usando guardia

2.0

El guardia comprueba una condición y, si es falso, entra en la rama. Las sucursales de los cheques de guardia deben dejar su bloque adjunto ya sea a través de `return` , `break` o `continue` (si corresponde); no hacerlo resulta en un error de compilación. Esto tiene la ventaja de que cuando se escribe una `guard` , no es posible dejar que el flujo continúe accidentalmente (como sería posible con un `if`).

El uso de guardas puede ayudar a mantener bajos los niveles de anidamiento , lo que generalmente mejora la legibilidad del código.

```
func printNum(num: Int) {
    guard num == 10 else {
        print("num is not 10")
        return
    }
    print("num is 10")
}
```

Guard también puede verificar si hay un valor en un opcional , y luego desenvolverlo en el alcance externo:

```
func printOptionalNum(num: Int?) {
    guard let unwrappedNum = num else {
        print("num does not exist")
        return
    }
    print(unwrappedNum)
}
```

Guard puede combinar el desenvolvimiento opcional y la verificación de condición usando la palabra clave `where` :

```
func printOptionalNum(num: Int?) {
guard let unwrappedNum = num, unwrappedNum == 10 else {
    print("num does not exist or is not 10")
    return
}
print(unwrappedNum)
}
```

Condicionales básicos: declaraciones if

Una **sentencia `if`** comprueba si una condición Bool es `true` :

```
let num = 10

if num == 10 {
    // Code inside this block only executes if the condition was true.
    print("num is 10")
}

let condition = num == 10   // condition's type is Bool
if condition {
    print("num is 10")
}
```

`if` declaraciones aceptan **`else if`** y **`else`** blocks, que pueden probar condiciones alternativas y proporcionar un respaldo:

```
let num = 10
if num < 10 {  // Execute the following code if the first condition is true.
    print("num is less than 10")
} else if num == 10 {  // Or, if not, check the next condition...
    print("num is 10")
} else {  // If all else fails...
    print("all other conditions were false, so num is greater than 10")
}
```

Operadores básicos como `&&` y `||` Puede ser utilizado para múltiples condiciones:

El operador lógico AND

```
let num = 10
let str = "Hi"
if num == 10 && str == "Hi" {
    print("num is 10, AND str is \"Hi\"")
}
```

Si `num == 10` era falso, el segundo valor no sería evaluado. Esto se conoce como evaluación de cortocircuito.

El operador lógico O

```
if num == 10 || str == "Hi" {
    print("num is 10, or str is \"Hi\")
}
```

Si `num == 10` es verdadero, el segundo valor no sería evaluado.

El operador lógico NO

```
if !str.isEmpty {
    print("str is not empty")
}
```

Encuadernación opcional y cláusulas "donde".

Los opcionales se deben *desenvolver* antes de que se puedan usar en la mayoría de las expresiones. `if let` es un *enlace opcional* , que tiene éxito si el valor opcional **no** era **`nil`** :

```
let num: Int? = 10 // or: let num: Int? = nil

if let unwrappedNum = num {
    // num has type Int?; unwrappedNum has type Int
    print("num was not nil: \(unwrappedNum + 1)")
} else {
    print("num was nil")
}
```

Puede reutilizar el **mismo nombre** para la nueva variable vinculada, sombreando el original:

```
// num originally has type Int?
if let num = num {
    // num has type Int inside this block
}
```

1.2 3.0

Combine múltiples enlaces opcionales con comas (`,`):

```
if let unwrappedNum = num, let unwrappedStr = str {
    // Do something with unwrappedNum & unwrappedStr
} else if let unwrappedNum = num {
    // Do something with unwrappedNum
} else {
    // num was nil
}
```

Aplique más restricciones después del enlace opcional utilizando una cláusula `where` :

```
if let unwrappedNum = num where unwrappedNum % 2 == 0 {
```

```
    print("num is non-nil, and it's an even number")
}
```

Si te sientes aventurero, intercala cualquier número de enlaces opcionales y cláusulas `where` :

```
if let num = num                           // num must be non-nil
    where num % 2 == 1,                    // num must be odd
    let str = str,                         // str must be non-nil
    let firstChar = str.characters.first   // str must also be non-empty
    where firstChar != "x"                 // the first character must not be "x"
{
    // all bindings & conditions succeeded!
}
```

3.0

En Swift 3, `where` se han reemplazado las cláusulas (SE-0099): simplemente use otra , para separar enlaces opcionales y condiciones booleanas.

```
if let unwrappedNum = num, unwrappedNum % 2 == 0 {
    print("num is non-nil, and it's an even number")
}

if let num = num,                           // num must be non-nil
    num % 2 == 1,                           // num must be odd
    let str = str,                          // str must be non-nil
    let firstChar = str.characters.first,   // str must also be non-empty
    firstChar != "x"                        // the first character must not be "x"
{
    // all bindings & conditions succeeded!
}
```

Operador ternario

Las condiciones también pueden evaluarse en una sola línea utilizando el operador ternario:

Si quisiera determinar el mínimo y el máximo de dos variables, podría usar sentencias if, así:

```
let a = 5
let b = 10
let min: Int

if a < b {
    min = a
} else {
    min = b
}

let max: Int

if a > b {
    max = a
} else {
    max = b
}
```

El operador condicional ternario toma una condición y devuelve uno de dos valores, dependiendo

de si la condición era verdadera o falsa. La sintaxis es la siguiente: Esto es equivalente a tener la expresión:

```
(<CONDITION>) ? <TRUE VALUE> : <FALSE VALUE>
```

El código anterior se puede reescribir usando el operador condicional ternario de la siguiente manera:

```
let a = 5
let b = 10
let min = a < b ? a : b
let max = a > b ? a : b
```

En el primer ejemplo, la condición es a <b. Si esto es cierto, el resultado asignado de nuevo a min será de a; si es falso, el resultado será el valor de b.

Nota: Debido a que encontrar el mayor o el menor de dos números es una operación tan común, la biblioteca estándar de Swift proporciona dos funciones para este propósito: max y min.

Operador sin coalescencia

El operador nulo coalescente <OPTIONAL> ?? <DEFAULT VALUE> desenvuelve el <OPTIONAL> si contiene un valor, o devuelve <DEFAULT VALUE> si es nulo. <OPTIONAL> es siempre de un tipo opcional. <DEFAULT VALUE> debe coincidir con el tipo que está almacenado dentro de <OPTIONAL> .

El operador de unión nula es una abreviatura del código siguiente que utiliza un operador ternario:

```
a != nil ? a! : b
```

Esto puede ser verificado por el siguiente código:

```
(a ?? b) == (a != nil ? a! : b) // ouputs true
```

Tiempo para un ejemplo

```
let defaultSpeed:String = "Slow"
var userEnteredSpeed:String? = nil

print(userEnteredSpeed ?? defaultSpeed) // ouputs "Slow"

userEnteredSpeed = "Fast"
print(userEnteredSpeed ?? defaultSpeed) // ouputs "Fast"
```

Lea Condicionales en línea: https://riptutorial.com/es/swift/topic/475/condicionales

Capítulo 16: Conjuntos

Examples

Declarar Conjuntos

Los conjuntos son colecciones desordenadas de valores únicos. Los valores únicos deben ser del mismo tipo.

```
var colors = Set<String>()
```

Puede declarar un conjunto con valores utilizando la sintaxis literal de la matriz.

```
var favoriteColors: Set<String> = ["Red", "Blue", "Green", "Blue"]
// {"Blue", "Green", "Red"}
```

Modificar valores en un conjunto.

```
var favoriteColors: Set = ["Red", "Blue", "Green"]
//favoriteColors = {"Blue", "Green", "Red"}
```

Puede utilizar el método de insert(_:) para agregar un nuevo elemento a un conjunto.

```
favoriteColors.insert("Orange")
//favoriteColors = {"Red", "Green", "Orange", "Blue"}
```

Puede usar el método remove(_:) para eliminar un elemento de un conjunto. Devuelve el valor que contiene opcional que se eliminó o nulo si el valor no estaba en el conjunto.

```
let removedColor = favoriteColors.remove("Red")
//favoriteColors = {"Green", "Orange", "Blue"}
// removedColor = Optional("Red")

let anotherRemovedColor = favoriteColors.remove("Black")
// anotherRemovedColor = nil
```

Comprobando si un conjunto contiene un valor

```
var favoriteColors: Set = ["Red", "Blue", "Green"]
//favoriteColors = {"Blue", "Green", "Red"}
```

Puede usar el método de contains(_:) para verificar si un conjunto contiene un valor. Devolverá verdadero si el conjunto contiene ese valor.

```
if favoriteColors.contains("Blue") {
    print("Who doesn't like blue!")
}
// Prints "Who doesn't like blue!"
```

Realización de operaciones en sets.

Valores comunes de ambos conjuntos:

Puede usar el método intersect(_:) para crear un nuevo conjunto que contenga todos los valores

comunes a ambos conjuntos.

```
let favoriteColors: Set = ["Red", "Blue", "Green"]
let newColors: Set = ["Purple", "Orange", "Green"]

let intersect = favoriteColors.intersect(newColors) // a AND b
// intersect = {"Green"}
```

Todos los valores de cada conjunto:

Puede usar el método de union(_:) para crear un nuevo conjunto que contenga todos los valores únicos de cada conjunto.

```
let union = favoriteColors.union(newColors) // a OR b
// union = {"Red", "Purple", "Green", "Orange", "Blue"}
```

Observe cómo el valor "Verde" solo aparece una vez en el nuevo conjunto.

Valores que no existen en ambos conjuntos:

Puede usar el método exclusiveOr(_:) para crear un nuevo conjunto que contenga los valores únicos de cualquiera de los dos conjuntos.

```
let exclusiveOr = favoriteColors.exclusiveOr(newColors) // a XOR b
// exclusiveOr = {"Red", "Purple", "Orange", "Blue"}
```

Observe cómo el valor "Verde" no aparece en el nuevo conjunto, ya que estaba en ambos conjuntos.

Valores que no están en un conjunto:

Puede usar el método subtract(_:) para crear un nuevo conjunto que contenga valores que no estén en un conjunto específico.

```
let subtract = favoriteColors.subtract(newColors) // a - (a AND b)
// subtract = {"Blue", "Red"}
```

Observe que el valor "Verde" no aparece en el nuevo conjunto, ya que también estaba en el segundo conjunto.

Agregando valores de mi propio tipo a un conjunto

Para definir un Set de su propio tipo, debe ajustar su tipo a Hashable

```
struct Starship: Hashable {
    let name: String
    var hashValue: Int { return name.hashValue }
}

func ==(left:Starship, right: Starship) -> Bool {
    return left.name == right.name
}
```

Ahora puedes crear un Set de Starship(s)

```
let ships : Set<Starship> = [Starship(name:"Enterprise D"), Starship(name:"Voyager"),
Starship(name:"Defiant") ]
```

CountedSet

3.0

Swift 3 presenta la clase CountedSet (es la versión Swift de la NSCountedSet Objective-C).

CountedSet, como lo sugiere el nombre, realiza un seguimiento de cuántas veces está presente un valor.

```
let countedSet = CountedSet()
countedSet.add(1)
countedSet.add(1)
countedSet.add(1)
countedSet.add(2)

countedSet.count(for: 1) // 3
countedSet.count(for: 2) // 1
```

Lea Conjuntos en línea: https://riptutorial.com/es/swift/topic/371/conjuntos

Capítulo 17: Control de acceso

Sintaxis

- Proyecto de clase privada
- let car = Car ("Ford", modelo: "Escape") // predeterminado interno
- Enumeración pública Género
- función privada calculaMarketCap ()
- anular la función interna setupView ()
- área de var privado (conjunto) = 0

Observaciones

1. ***Observación básica:***

A continuación se muestran los tres niveles de acceso desde el acceso más alto (menos restrictivo) al acceso más bajo (más restrictivo)

El acceso **público** permite acceder a clases, estructuras, variables, etc. desde cualquier archivo dentro del modelo, pero más importante fuera del módulo si el archivo externo importa el módulo que contiene el código de acceso público. Es popular utilizar el acceso público al crear un marco.

El acceso **interno** permite que los archivos solo con el módulo de las entidades utilicen las entidades. Todas las entidades tienen **un** nivel de acceso **interno** por defecto (con algunas excepciones).

El acceso **privado** evita que la entidad se use fuera de ese archivo.

2. ***Observación de subclases:***

Una subclase no puede tener un acceso más alto que su superclase.

3. ***Observación de Getter & Setter:***

Si el establecedor de la propiedad es privado, el captador es interno (que es el predeterminado). También puede asignar un nivel de acceso para el getter y el setter. Estos principios también se aplican a los *subíndices* también

4. ***Observación General:***

Otros tipos de entidades incluyen: Inicializadores, Protocolos, Extensiones, Genéricos y Alias de Tipo

Examples

Ejemplo básico usando un Struct

3.0

En Swift 3 hay múltiples niveles de acceso. Este ejemplo los usa a todos excepto para open :

```
public struct Car {

    public let make: String
    let model: String //Optional keyword: will automatically be "internal"
    private let fullName: String
    fileprivate var otherName: String
```

```
    public init(_ make: String, model: String) {
        self.make = make
        self.model = model
        self.fullName = "\(make)\(model)"
        self.otherName = "\(model) - \(make)"
    }
}
```

Supongamos que myCar se inicializó así:

```
let myCar = Car("Apple", model: "iCar")
```

Car.make (público)

```
print(myCar.make)
```

Esta impresión funcionará en todas partes, incluidos los objetivos que importan Car .

Car.model (interno)

```
print(myCar.model)
```

Esto se compilará si el código está en el mismo objetivo que Car .

Car.otherName (fileprivate)

```
print(myCar.otherName)
```

Esto solo funcionará si el código está *en el mismo archivo* que Car .

Car.fullName (privado)

```
print(myCar.fullName)
```

Esto no funcionará en Swift 3. Las propiedades private solo se pueden acceder dentro de la misma struct / class .

```
public struct Car {

    public let make: String       //public
    let model: String             //internal
    private let fullName: String! //private

    public init(_ make: String, model model: String) {
        self.make = make
        self.model = model
        self.fullName = "\(make)\(model)"
    }
}
```

Si la entidad tiene múltiples niveles de acceso asociados, Swift busca el nivel de acceso más bajo. Si existe una variable privada en una clase pública, la variable aún se considerará privada.

Ejemplo de subclasificación

```
public class SuperClass {
    private func secretMethod() {}
}

internal class SubClass: SuperClass {
    override internal func secretMethod() {
        super.secretMethod()
    }
}
```

Getters and Setters Ejemplo

```
struct Square {
    private(set) var area = 0

    var side: Int = 0 {
        didSet {
            area = side*side
        }
    }
}

public struct Square {
    public private(set) var area = 0
    public var side: Int = 0 {
        didSet {
            area = side*side
        }
    }
    public init() {}
}
```

Lea Control de acceso en línea: https://riptutorial.com/es/swift/topic/1075/control-de-acceso

Capítulo 18: Controlador de finalización

Introducción

Prácticamente todas las aplicaciones utilizan funciones asíncronas para evitar que el código bloquee el subproceso principal.

Examples

Controlador de finalización sin argumento de entrada

```
func sampleWithCompletion(completion:@escaping (()-> ())){
    let delayInSeconds = 1.0
    DispatchQueue.main.asyncAfter(deadline: DispatchTime.now() + delayInSeconds) {

        completion()

    }
}

//Call the function
sampleWithCompletion {
    print("after one second")
}
```

Controlador de finalización con argumento de entrada

```
enum ReadResult{
    case Successful
    case Failed
    case Pending
}

struct OutpuData {
    var data = Data()
    var result: ReadResult
    var error: Error?
}

func readData(from url: String, completion: @escaping (OutpuData) ->  Void) {
    var _data = OutpuData(data: Data(), result: .Pending, error: nil)
    DispatchQueue.global().async {
        let url=URL(string: url)
        do {
            let rawData = try Data(contentsOf: url!)
            _data.result = .Successful
            _data.data = rawData
            completion(_data)
        }
        catch let error {
            _data.result = .Failed
            _data.error = error
            completion(_data)
        }

    }
}
```

```
readData(from: "https://raw.githubusercontent.com/trev/bearcal/master/sample-data-large.json")
{ (output) in
    switch output.result {
    case .Successful:
        break
    case .Failed:
        break
    case .Pending:
        break

    }
}
```

Lea Controlador de finalización en línea:
https://riptutorial.com/es/swift/topic/9378/controlador-de-finalizacion

Capítulo 19: Convenciones de estilo

Observaciones

Swift tiene una guía de estilo oficial: Swift.org API Design Guidelines . Otra guía popular es The Official raywenderlich.com Swift Style Guide.

Examples

Claro uso

Evitar la ambigüedad

El nombre de las clases, estructuras, funciones y variables debe evitar la ambigüedad.

Ejemplo:

```
extension List {
    public mutating func remove(at position: Index) -> Element {
        // implementation
    }
}
```

La llamada a la función a esta función se verá así:

```
list.remove(at: 42)
```

De esta manera, se evita la ambigüedad. Si la llamada a la función sería simplemente list.remove(42) no estaría claro, si se eliminaría un Elemento igual a 42 o si se eliminaría el Elemento en el Índice 42.

Evitar la redundancia

El nombre de las funciones no debe contener información redundante.

Un mal ejemplo sería:

```
extension List {
    public mutating func removeElement(element: Element) -> Element? {
        // implementation
    }
}
```

Una llamada a la función puede verse como list.removeElement(someObject) . La variable someObject ya indica que se ha eliminado un elemento. Sería mejor que la firma de función se vea así:

```
extension List {
    public mutating func remove(_ member: Element) -> Element? {
        // implementation
    }
}
```

La llamada a esta función se ve así: list.remove(someObject) .

Nombrando variables de acuerdo a su rol.

Las variables deben ser nombradas por su rol (por ejemplo, proveedor, saludo) en lugar de su tipo (por ejemplo, fábrica, cadena, etc.)

Alto acoplamiento entre nombre de protocolo y nombres de variable

Si el nombre del tipo describe su rol en la mayoría de los casos (por ejemplo, Iterator), el tipo debe nombrarse con el sufijo `Type`. (por ejemplo, IteratorType)

Proporcionar detalles adicionales cuando se utilizan parámetros de tipo débil

Si el tipo de un objeto no indica claramente su uso en una llamada de función, la función debe nombrarse con un nombre precedente para cada parámetro de tipo débil, describiendo su uso.
Ejemplo:

```
func addObserver(_ observer: NSObject, forKeyPath path: String)
```

a la que se vería una llamada como `object.addObserver (self, forKeyPath: path)

en lugar de

```
func add(_ observer: NSObject, for keyPath: String)
```

a la que una llamada se vería como object.add(self, for: path)

Uso fluido

Usando lenguaje natural

Las llamadas de funciones deben estar cerca del idioma inglés natural.

Ejemplo:

```
list.insert(element, at: index)
```

en lugar de

```
list.insert(element, position: index)
```

Métodos de fábrica de nombres

Los métodos de fábrica deben comenzar con el prefijo `make`.

Ejemplo:

```
factory.makeObject()
```

Nombrando Parámetros en Inicializadores y Métodos de Fábrica

El nombre del primer argumento no debe involucrarse al nombrar un método de fábrica o un inicializador.

Ejemplo:

```
factory.makeObject(key: value)
```

En lugar de:

```
factory.makeObject(havingProperty: value)
```

Nombrar de acuerdo a los efectos secundarios

- Las funciones con efectos secundarios (funciones de mutación) deben nombrarse usando verbos o sustantivos con el prefijo form- .
- Las funciones sin efectos secundarios (funciones no mutantes) deben nombrarse usando sustantivos o verbos con el sufijo -ing o -ed .

Ejemplo: funciones mutantes:

```
print(value)
array.sort()                  // in place sorting
list.add(value)               // mutates list
set.formUnion(anotherSet)     // set is now the union of set and anotherSet
```

Funciones no mutantes:

```
let sortedArray = array.sorted()     // out of place sorting
let union = set.union(anotherSet)    // union is now the union of set and another set
```

Funciones o variables booleanas

Las declaraciones que involucran booleanos deben leerse como afirmaciones.

Ejemplo:

```
set.isEmpty
line.intersects(anotherLine)
```

Protocolos de denominación

- Los protocolos que describen qué es algo se deben nombrar usando sustantivos.
- Los protocolos que describen las capacidades deben tener -able , -ible o -ing como sufijo.

Ejemplo:

```
Collection        // describes that something is a collection
ProgressReporting // describes that something has the capability of reporting progress
Equatable         // describes that something has the capability of being equal to something
```

Tipos y propiedades

Tipos, variables y propiedades deben leerse como sustantivos.

Ejemplo:

```
let factory = ...
```

```
let list = [1, 2, 3, 4]
```

Capitalización

Tipos y Protocolos

Los nombres de tipo y protocolo deben comenzar con una letra mayúscula.

Ejemplo:

```
protocol Collection {}
struct String {}
class UIView {}
struct Int {}
enum Color {}
```

Todo lo demas...

Las variables, constantes, funciones y casos de enumeración deben comenzar con una letra minúscula.

Ejemplo:

```
let greeting = "Hello"
let height = 42.0

enum Color {
    case red
    case green
    case blue
}

func print(_ string: String) {
    ...
}
```

El caso de Carmel:

Todos los nombres deben utilizar el caso de camello apropiado. Estuche de camello superior para nombres de tipo / protocolo y estuche de camello inferior para todo lo demás.

Estuche de camello superior:

```
protocol IteratorType { ... }
```

Funda de camello inferior:

```
let inputView = ...
```

Abreviaturas

Deben evitarse las abreviaturas a menos que se usen comúnmente (por ejemplo, URL, ID). Si se usa una abreviatura, todas las letras deben tener el mismo caso.

Ejemplo:

```
let userID: UserID = ...
let urlString: URLString = ...
```

Lea Convenciones de estilo en línea: https://riptutorial.com/es/swift/topic/3031/convenciones-de-estilo

Capítulo 20: Cuerdas y personajes

Sintaxis

- String.characters // devuelve una matriz de los caracteres en la cadena
- String.characters.count // Devuelve el número de caracteres
- String.utf8 // A String.UTF8View, devuelve los puntos de caracteres UTF-8 en la cadena
- String.utf16 // A String.UTF16View, devuelve los puntos de caracteres UTF-16 en la cadena
- String.unicodeScalars // A String.UnicodeScalarView, devuelve los puntos de caracteres UTF-32 en la cadena
- String.isEmpty // Devuelve true si la cadena no contiene ningún texto
- String.hasPrefix (String) // Devuelve true si la cadena tiene un prefijo con el argumento
- String.hasSuffix (String) // Devuelve true si String tiene el sufijo del argumento
- String.startIndex // Devuelve el índice que corresponde al primer carácter de la cadena
- String.endIndex // Devuelve el índice que corresponde al punto después del último carácter en la cadena
- String.components (SeparadoBy: String) // Devuelve una matriz que contiene las subcadenas separadas por la cadena separadora dada
- String.append (Character) // Agrega el carácter (dado como argumento) a la cadena

Observaciones

Una String en Swift es una colección de caracteres y, por extensión, una colección de escalares Unicode. Debido a que las cadenas Swift se basan en Unicode, pueden ser cualquier valor escalar de Unicode, incluidos los idiomas distintos del inglés y los emojis.

Debido a que dos escalares podrían combinarse para formar un solo carácter, el número de escalares en una Cadena no es necesariamente siempre igual al número de caracteres.

Para obtener más información sobre las cadenas, consulte El lenguaje de programación Swift y la Referencia de la estructura de cadenas .

Para detalles de implementación, vea "Swift String Design"

Examples

Literales de cuerdas y personajes

Los literales de cadena en Swift están delimitados con comillas dobles ("):

```
let greeting = "Hello!"  // greeting's type is String
```

Los caracteres se pueden inicializar a partir de literales de cadena, siempre que el literal contenga solo un grupo de grafemas:

```
let chr: Character = "H" // valid
let chr2: Character = "" // valid
let chr3: Character = "abc" // invalid - multiple grapheme clusters
```

Interpolación de cuerdas

La interpolación de cadenas permite inyectar una expresión directamente en un literal de cadena. Esto se puede hacer con todos los tipos de valores, incluidas cadenas, enteros, números de punto flotante y más.

La sintaxis es una barra invertida seguida de paréntesis que envuelven el valor: \(value) . Cualquier expresión válida puede aparecer entre paréntesis, incluidas las llamadas a funciones.

```
let number = 5
let interpolatedNumber = "\(number)"  // string is "5"
let fortyTwo = "\(6 * 7)"             // string is "42"

let example = "This post has \(number) view\(number == 1 ? "" : "s")"
// It will output "This post has 5 views" for the above example.
// If the variable number had the value 1, it would output "This post has 1 view" instead.
```

Para los tipos personalizados, el comportamiento predeterminado de la interpolación de cadenas es que "\(myobj)" es equivalente a String(myobj) , la misma representación utilizada por print(myobj) . Puede personalizar este comportamiento implementando el protocolo CustomStringConvertible para su tipo.

3.0

Para Swift 3, de acuerdo con SE-0089 , String.init<T>(_:) ha sido renombrada a String.init<T>(describing:) .

La interpolación de cadena "\(myobj)" preferirá el nuevo String.init<T: LosslessStringConvertible>(_:) , pero volverá a init<T>(describing:) si el valor no es LosslessStringConvertible .

Caracteres especiales

Ciertos caracteres requieren una **secuencia de escape** especial para usarlos en literales de cadena:

Personaje	Sentido
\0	el caracter nulo
\\	una barra invertida plana, \
\t	un carácter de tabulación
\v	una pestaña vertical
\r	un retorno de carro
\n	un salto de línea ("nueva línea")
\"	una cita doble, "
\'	una comilla, '
\u{n}	El punto de código Unicode *n* (en hexadecimal)

Ejemplo:

```
let message = "Then he said, \"I \u{1F496} you!\""

print(message) // Then he said, "I  you!"
```

Cuerdas de concatenación

Concatene cadenas con el operador + para producir una nueva cadena:

```
let name = "John"
let surname = "Appleseed"
```

```
let fullName = name + " " + surname  // fullName is "John Appleseed"
```

Agregue a una cadena **mutable** usando el operador de asignación compuesto += , o usando un método:

```
let str2 = "there"
var instruction = "look over"
instruction += " " + str2  // instruction is now "look over there"

var instruction = "look over"
instruction.append(" " + str2)  // instruction is now "look over there"
```

Agregue un solo carácter a una cadena mutable:

```
var greeting: String = "Hello"
let exclamationMark: Character = "!"
greeting.append(exclamationMark)
// produces a modified String (greeting) = "Hello!"
```

Agregar varios caracteres a una cadena mutable

```
var alphabet:String = "my ABCs: "
alphabet.append(contentsOf: (0x61...0x7A).map(UnicodeScalar.init)
                                          .map(Character.init) )
// produces a modified string (alphabet) = "my ABCs: abcdefghijklmnopqrstuvwxyz"
```

3.0

appendContentsOf(_:) ha sido renombrado a append(_:) .

Unir una secuencia de cadenas para formar una nueva cadena utilizando joinWithSeparator(_:) :

```
let words = ["apple", "orange", "banana"]
let str = words.joinWithSeparator(" & ")

print(str)   // "apple & orange & banana"
```

3.0

joinWithSeparator(_:) ha cambiado de nombre a joinWithSeparator(_:) joined(separator:) .

El separator es la cadena vacía de forma predeterminada, por lo que ["a", "b", "c"].joined() == "abc" .

Examina y compara cuerdas

Compruebe si una cadena está vacía:

```
if str.isEmpty {
    // do something if the string is empty
}

// If the string is empty, replace it with a fallback:
let result = str.isEmpty ? "fallback string" : str
```

Compruebe si dos cadenas son iguales (en el sentido de equivalencia canónica de Unicode):

```
"abc" == "def"          // false
"abc" == "ABC"          // false
"abc" == "abc"          // true
```

```
// "LATIN SMALL LETTER A WITH ACUTE" == "LATIN SMALL LETTER A" + "COMBINING ACUTE ACCENT"
"\u{e1}" == "a\u{301}"  // true
```

Compruebe si una cadena comienza / termina con otra cadena:

```
"fortitude".hasPrefix("fort")      // true
"Swift Language".hasSuffix("age")  // true
```

Codificación y descomposición de cadenas

Una cadena Swift está hecha de puntos de código Unicode . Se puede descomponer y codificar de diferentes maneras.

```
let str = "ที่👌①!"
```

Cuerdas en descomposición

Los characters una cadena son agrupamientos de grafemas extendidos Unicode:

```
Array(str.characters)  // ["ที่", "👌", "①", "!"]
```

Los unicodeScalars son los puntos de código Unicode que forman una cadena (observe que ที่ es un grupo de grafemas, pero 3 puntos de código - 3607, 3637, 3656 - por lo que la longitud de la matriz resultante no es la misma que con los characters):

```
str.unicodeScalars.map{ $0.value }  // [3607, 3637, 3656, 128076, 9312, 33]
```

Puede codificar y descomponer cadenas como UTF-8 (una secuencia de UInt8 s) o UTF-16 (una secuencia de UInt16 s):

```
Array(str.utf8)   // [224, 184, 151, 224, 184, 181, 224, 185, 136, 240, 159, 145, 140, 226,
145, 160, 33]
Array(str.utf16)  // [3607, 3637, 3656, 55357, 56396, 9312, 33]
```

Longitud de la cuerda y la iteración

Los characters una cadena, unicodeScalars , utf8 y utf16 son todos Collection s, por lo que puede obtener su count e iterar sobre ellos:

```
// NOTE: These operations are NOT necessarily fast/cheap!

str.characters.count     // 4
str.unicodeScalars.count // 6
str.utf8.count           // 17
str.utf16.count          // 7
```

```
for c in str.characters { // ...
for u in str.unicodeScalars { // ...
for byte in str.utf8 { // ...
for byte in str.utf16 { // ...
```

Unicode

Estableciendo valores

Usando Unicode directamente

```
var str: String = "I want to visit □□, Москва, मुंबई, القاهرة, and □□□. □"
var character: Character = "□"
```

Usando valores hexadecimales

```
var str: String = "\u{61}\u{5927}\u{1F34E}\u{3C0}" // a□□π
var character: Character = "\u{65}\u{301}" // é = "e" + accent mark
```

Tenga en cuenta que el Character Swift puede estar compuesto por varios puntos de código Unicode, pero parece ser un solo carácter. Esto se llama un Clúster Grapheme Extendido.

Conversiones

Cuerda -> Hexagonal

```
// Accesses views of different Unicode encodings of `str`
str.utf8
str.utf16
str.unicodeScalars // UTF-32
```

Hexagonal -> Cuerda

```
let value0: UInt8 = 0x61
let value1: UInt16 = 0x5927
let value2: UInt32 = 0x1F34E

let string0 = String(UnicodeScalar(value0)) // a
let string1 = String(UnicodeScalar(value1)) // □
let string2 = String(UnicodeScalar(value2)) // □

// convert hex array to String
let myHexArray = [0x43, 0x61, 0x74, 0x203C, 0x1F431] // an Int array
var myString = ""
for hexValue in myHexArray {
    myString.append(UnicodeScalar(hexValue))
}
print(myString) // Cat‼□
```

Tenga en cuenta que para UTF-8 y UTF-16, la conversión no siempre es tan fácil porque cosas como emoji no pueden codificarse con un solo valor UTF-16. Se necesita una pareja sustituta.

Cuerdas de inversión

2.2

```
let aString = "This is a test string."

// first, reverse the String's characters
let reversedCharacters = aString.characters.reverse()

// then convert back to a String with the String() initializer
let reversedString = String(reversedCharacters)
```

```
print(reversedString) // ".gnirts tset a si sihT"
```

3.0

```
let reversedCharacters = aString.characters.reversed()
let reversedString = String(reversedCharacters)
```

Cadenas mayúsculas y minúsculas

Para hacer todos los caracteres en una cadena mayúscula o minúscula:

2.2

```
let text = "AaBbCc"
let uppercase = text.uppercaseString // "AABBCC"
let lowercase = text.lowercaseString // "aabbcc"
```

3.0

```
let text = "AaBbCc"
let uppercase = text.uppercased() // "AABBCC"
let lowercase = text.lowercased() // "aabbcc"
```

Compruebe si la cadena contiene caracteres de un conjunto definido

Letras

3.0

```
let letters = CharacterSet.letters

let phrase = "Test case"
let range = phrase.rangeOfCharacter(from: letters)

// range will be nil if no letters is found
if let test = range {
   print("letters found")
}
else {
   print("letters not found")
}
```

2.2

```
let letters = NSCharacterSet.letterCharacterSet()

let phrase = "Test case"
let range = phrase.rangeOfCharacterFromSet(letters)

// range will be nil if no letters is found
if let test = range {
   print("letters found")
}
else {
  print("letters not found")
}
```

La nueva estructura CharacterSet que también está NSCharacterSet clase Objective-C

NSCharacterSet define varios conjuntos predefinidos como:

- decimalDigits
  - capitalizedLetters
  - alphanumerics
  - controlCharacters
  - illegalCharacters
  - Y más puede encontrar en la referencia NSCharacterSet .

También puedes definir tu propio conjunto de caracteres:

3.0

```
let phrase = "Test case"
let charset = CharacterSet(charactersIn: "t")

if let _ = phrase.rangeOfCharacter(from: charset, options: .caseInsensitive) {
   print("yes")
}
else {
   print("no")
}
```

2.2

```
let charset = NSCharacterSet(charactersInString: "t")

if let _ = phrase.rangeOfCharacterFromSet(charset, options: .CaseInsensitiveSearch, range:
nil) {
   print("yes")
}
else {
    print("no")
}
```

También puede incluir rango:

3.0

```
let phrase = "Test case"
let charset = CharacterSet(charactersIn: "t")

if let _ = phrase.rangeOfCharacter(from: charset, options: .caseInsensitive, range:
phrase.startIndex..<phrase.endIndex)) {
   print("yes")
}
else {
   print("no")
}
```

Contar las ocurrencias de un personaje en una cadena

Dada una String y un Character

```
let text = "Hello World"
let char: Character = "o"
```

Podemos contar el número de veces que el Character aparece en la String usando

```
let sensitiveCount = text.characters.filter { $0 == char }.count // case-sensitive
let insensitiveCount = text.lowercaseString.characters.filter { $0 ==
Character(String(char).lowercaseString) } // case-insensitive
```

Eliminar caracteres de una cadena no definida en Set

2.2

```
func removeCharactersNotInSetFromText(text: String, set: Set<Character>) -> String {
   return String(text.characters.filter { set.contains( $0) })
}

let text = "Swift 3.0 Come Out"
var chars =
Set([Character]("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLKMNOPQRSTUVWXYZ".characters))
let newText = removeCharactersNotInSetFromText(text, set: chars) // "SwiftComeOut"
```

3.0

```
func removeCharactersNotInSetFromText(text: String, set: Set<Character>) -> String {
  return String(text.characters.filter { set.contains( $0) })
}

let text = "Swift 3.0 Come Out"
var chars =
Set([Character]("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLKMNOPQRSTUVWXYZ".characters))
let newText = removeCharactersNotInSetFromText(text: text, set: chars)
```

Formato de cadenas

Ceros a la izquierda

```
let number: Int = 7
let str1 = String(format: "%03d", number) // 007
let str2 = String(format: "%05d", number) // 00007
```

Números despues de decimal

```
let number: Float = 3.14159
let str1 = String(format: "%.2f", number) // 3.14
let str2 = String(format: "%.4f", number) // 3.1416 (rounded)
```

Decimal a hexadecimal

```
let number: Int = 13627
let str1 = String(format: "%2X", number) // 353B
let str2 = String(format: "%2x", number) // 353b (notice the lowercase b)
```

Alternativamente, se podría usar un inicializador especializado que haga lo mismo:

```
let number: Int = 13627
let str1 = String(number, radix: 16, uppercase: true) //353B
let str2 = String(number, radix: 16) // 353b
```

Decimal a un número con radix arbitrario

```
let number: Int = 13627
let str1 = String(number, radix: 36) // aij
```

Radix es Int en [2, 36] .

Convertir una cadena Swift a un tipo de número

```
Int("123") // Returns 123 of Int type
Int("abcd") // Returns nil
Int("10") // Returns 10 of Int type
Int("10", radix: 2) // Returns 2 of Int type
Double("1.5") // Returns 1.5 of Double type
Double("abcd") // Returns nil
```

Tenga en cuenta que al hacer esto, se devuelve un valor Optional , que debe ser desempaquetado en consecuencia antes de ser utilizado.

Iteración de cuerdas

3.0

```
let string = "My fantastic string"
var index = string.startIndex

while index != string.endIndex {
    print(string[index])
    index = index.successor()
}
```

Nota: endIndex está después del final de la cadena (es decir, la string[string.endIndex] es un error, pero la string[string.startIndex] está bien). Además, en una cadena vacía (""), string.startIndex == string.endIndex es true . Asegúrese de verificar si hay cadenas vacías, ya que no puede llamar a startIndex.successor() en una cadena vacía.

3.0

En Swift 3, los índices de cadena ya no tienen successor() , predecessor() , advancedBy(_:) , advancedBy(_:limit:) o distanceTo(_:) .

En cambio, esas operaciones se mueven a la colección, que ahora es responsable de aumentar y disminuir sus índices.

Los métodos disponibles son .index(after:) , .index(before:) y .index(_:, offsetBy:) .

```
let string = "My fantastic string"
var currentIndex = string.startIndex

while currentIndex != string.endIndex {
    print(string[currentIndex])
    currentIndex = string.index(after: currentIndex)
}
```

Nota: estamos usando currentIndex como nombre de variable para evitar confusiones con el método .index .

Y, por ejemplo, si quieres ir por el otro lado:

3.0

```
var index:String.Index? = string.endIndex.predecessor()
```

```
while index != nil {
    print(string[index!])
    if index != string.startIndex {
        index = index.predecessor()
    }
    else {
        index = nil
    }
}
```

(O simplemente puedes revertir la cadena primero, pero si no necesitas recorrer toda la cadena, probablemente preferirías un método como este)

3.0

```
var currentIndex: String.Index? = string.index(before: string.endIndex)

while currentIndex != nil {
    print(string[currentIndex!])
    if currentIndex != string.startIndex {
        currentIndex = string.index(before: currentIndex!)
    }
    else {
        currentIndex = nil
    }
}
```

Tenga en cuenta que el Index es un tipo de objeto y no un Int . No puede acceder a un carácter de cadena de la siguiente manera:

```
let string = "My string"
string[2] // can't do this
string.characters[2] // and also can't do this
```

Pero puede obtener un índice específico de la siguiente manera:

3.0

```
index = string.startIndex.advanceBy(2)
```

3.0

```
currentIndex = string.index(string.startIndex, offsetBy: 2)
```

Y puede ir hacia atrás así:

3.0

```
index = string.endIndex.advancedBy(-2)
```

3.0

```
currentIndex = string.index(string.endIndex, offsetBy: -2)
```

Si puede exceder los límites de la cadena, o si desea especificar un límite, puede usar:

3.0

```
index = string.startIndex.advanceBy(20, limit: string.endIndex)
```

3.0

```
currentIndex = string.index(string.startIndex, offsetBy: 20, limitedBy: string.endIndex)
```

Alternativamente, uno solo puede iterar a través de los caracteres en una cadena, pero esto podría ser menos útil dependiendo del contexto:

```
for c in string.characters {
    print(c)
}
```

Eliminar WhiteSpace y NewLine iniciales y finales

3.0

```
let someString = "  Swift Language  \n"
let trimmedString =
someString.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceAndNewlineCharacterSet())
// "Swift Language"
```

El método stringByTrimmingCharactersInSet devuelve una nueva cadena creada al eliminar de ambos extremos los caracteres de cadena contenidos en un conjunto de caracteres determinado.

También podemos eliminar solo espacios en blanco o nueva línea.

Eliminando solo espacios en blanco:

```
let trimmedWhiteSpace =
someString.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceCharacterSet())
// "Swift Language  \n"
```

Eliminando solo nueva linea:

```
let trimmedNewLine =
someString.stringByTrimmingCharactersInSet(NSCharacterSet.newlineCharacterSet())
// "  Swift Language  "
```

3.0

```
let someString = "  Swift Language  \n"

let trimmedString = someString.trimmingCharacters(in: .whitespacesAndNewlines)
// "Swift Language"

let trimmedWhiteSpace = someString.trimmingCharacters(in: .whitespaces)
// "Swift Language  \n"

let trimmedNewLine = someString.trimmingCharacters(in: .newlines)
// "  Swift Language  "
```

Nota: todos estos métodos pertenecen a la Foundation . Use la import Foundation si la Fundación ya no se ha importado a través de otras bibliotecas como Cocoa o UIKit.

Convertir cadena a / desde datos / NSData

Para convertir String ay desde Data / NSData necesitamos codificar esta cadena con una codificación específica. El más famoso es UTF-8 que es una representación de 8 bits de caracteres Unicode, adecuada para la transmisión o el almacenamiento mediante sistemas basados en ASCII. Aquí está una lista de todas las String Encodings disponibles

String a Data / NSData

3.0

```
let data = string.data(using: .utf8)
```

2.2

```
let data = string.dataUsingEncoding(NSUTF8StringEncoding)
```

Data / NSData a String

3.0

```
let string = String(data: data, encoding: .utf8)
```

2.2

```
let string = String(data: data, encoding: NSUTF8StringEncoding)
```

Dividir una cadena en una matriz

En Swift, puedes separar fácilmente una Cadena en una matriz cortándola en un determinado carácter:

3.0

```
let startDate = "23:51"

let startDateAsArray = startDate.components(separatedBy: ":") // ["23", "51"]`
```

2.2

```
let startDate = "23:51"

let startArray = startDate.componentsSeparatedByString(":") // ["23", "51"]`
```

O cuando el separador no está presente:

3.0

```
let myText = "MyText"

let myTextArray = myText.components(separatedBy: " ") // myTextArray is ["MyText"]
```

2.2

```
let myText = "MyText"

let myTextArray = myText.componentsSeparatedByString(" ") // myTextArray is ["MyText"]
```

Lea Cuerdas y personajes en línea: https://riptutorial.com/es/swift/topic/320/cuerdas-y-

personajes

Capítulo 21: Derivación de clave PBKDF2

Examples

Clave basada en contraseña Derivación 2 (Swift 3)

La derivación de clave basada en contraseña se puede utilizar tanto para derivar una clave de cifrado a partir de un texto de contraseña como para guardar una contraseña con fines de autenticación.

Hay varios algoritmos hash que se pueden usar, incluidos SHA1, SHA256, SHA512, que se proporcionan en este código de ejemplo.

El parámetro de rondas se usa para hacer que el cálculo sea lento, de modo que un atacante tenga que dedicar un tiempo considerable a cada intento. Los valores de retardo típicos caen en los 100 ms a 500 ms, se pueden usar valores más cortos si hay un rendimiento inaceptable.

Este ejemplo requiere Crypto común
Es necesario tener un encabezado puente al proyecto:
#import <CommonCrypto/CommonCrypto.h>
Agregue el Security.framework al proyecto.

Parámetros:

```
password     password String
salt         salt Data
keyByteCount number of key bytes to generate
rounds       Iteration rounds

returns      Derived key


func pbkdf2SHA1(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data? {
    return pbkdf2(hash:CCPBKDFAlgorithm(kCCPRFHmacAlgSHA1), password:password, salt:salt,
keyByteCount:keyByteCount, rounds:rounds)
}

func pbkdf2SHA256(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data? {
    return pbkdf2(hash:CCPBKDFAlgorithm(kCCPRFHmacAlgSHA256), password:password, salt:salt,
keyByteCount:keyByteCount, rounds:rounds)
}

func pbkdf2SHA512(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data? {
    return pbkdf2(hash:CCPBKDFAlgorithm(kCCPRFHmacAlgSHA512), password:password, salt:salt,
keyByteCount:keyByteCount, rounds:rounds)
}

func pbkdf2(hash :CCPBKDFAlgorithm, password: String, salt: Data, keyByteCount: Int, rounds:
Int) -> Data? {
    let passwordData = password.data(using:String.Encoding.utf8)!
    var derivedKeyData = Data(repeating:0, count:keyByteCount)

    let derivationStatus = derivedKeyData.withUnsafeMutableBytes {derivedKeyBytes in
        salt.withUnsafeBytes { saltBytes in

            CCKeyDerivationPBKDF(
                CCPBKDFAlgorithm(kCCPBKDF2),
                password, passwordData.count,
                saltBytes, salt.count,
                hash,
                UInt32(rounds),
```

```
                derivedKeyBytes, derivedKeyData.count)
        }
    }
    if (derivationStatus != 0) {
        print("Error: \(derivationStatus)")
        return nil;
    }

    return derivedKeyData
}
```

Ejemplo de uso:

```
let password     = "password"
//let salt       = "saltData".data(using: String.Encoding.utf8)!
let salt         = Data(bytes: [0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let keyByteCount = 16
let rounds       = 100000

let derivedKey = pbkdf2SHA1(password:password, salt:salt, keyByteCount:keyByteCount,
rounds:rounds)
print("derivedKey (SHA1): \(derivedKey! as NSData)")
```

Ejemplo de salida:

```
derivedKey (SHA1): <6b9d4fa3 0385d128 f6d196ee 3f1d6dbf>
```

Clave basada en contraseña Derivación 2 (Swift 2.3)

Vea el ejemplo de Swift 3 para información de uso y notas

```
func pbkdf2SHA1(password: String, salt: [UInt8], keyCount: Int, rounds: Int) -> [UInt8]? {
    return pbkdf2(CCPBKDFAlgorithm(kCCPRFHmacAlgSHA1), password:password, salt:salt,
keyCount:keyCount, rounds:UInt32(rounds))
}

func pbkdf2SHA256(password: String, salt: [UInt8], keyCount: Int, rounds: Int) -> [UInt8]? {
    return pbkdf2(CCPBKDFAlgorithm(kCCPRFHmacAlgSHA256), password:password, salt:salt,
keyCount:keyCount, rounds:UInt32(rounds))
}

func pbkdf2SHA512(password: String, salt: [UInt8], keyCount: Int, rounds: Int) -> [UInt8]? {
    return pbkdf2(CCPBKDFAlgorithm(kCCPRFHmacAlgSHA512), password:password, salt:salt,
keyCount:keyCount, rounds:UInt32(rounds))
}

func pbkdf2(hash :CCPBKDFAlgorithm, password: String, salt: [UInt8], keyCount: Int, rounds:
UInt32!) -> [UInt8]! {
    let derivedKey   = [UInt8](count:keyCount, repeatedValue:0)
    let passwordData = password.dataUsingEncoding(NSUTF8StringEncoding)!

    let derivationStatus = CCKeyDerivationPBKDF(
        CCPBKDFAlgorithm(kCCPBKDF2),
        UnsafePointer<Int8>(passwordData.bytes), passwordData.length,
        UnsafePointer<UInt8>(salt), salt.count,
        CCPseudoRandomAlgorithm(hash),
        rounds,
        UnsafeMutablePointer<UInt8>(derivedKey),
        derivedKey.count)
```

```
    if (derivationStatus != 0) {
        print("Error: \(derivationStatus)")
        return nil;
    }

    return derivedKey
}
```

Ejemplo de uso:

```
let password = "password"
// let salt  = [UInt8]("saltData".utf8)
let salt     = [UInt8]([0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let rounds   = 100_000
let keyCount = 16

let derivedKey = pbkdf2SHA1(password, salt:salt, keyCount:keyCount, rounds:rounds)
print("derivedKey (SHA1):   \(NSData(bytes:derivedKey!, length:derivedKey!.count))")
```

Ejemplo de salida:

```
derivedKey (SHA1):   <6b9d4fa3 0385d128 f6d196ee 3f1d6dbf>
```

Calibración de derivación de clave basada en contraseña (Swift 2.3)

Vea el ejemplo de Swift 3 para información de uso y notas

```
func pbkdf2SHA1Calibrate(password:String, salt:[UInt8], msec:Int) -> UInt32 {
    let actualRoundCount: UInt32 = CCCalibratePBKDF(
        CCPBKDFAlgorithm(kCCPBKDF2),
        password.utf8.count,
        salt.count,
        CCPseudoRandomAlgorithm(kCCPRFHmacAlgSHA1),
        kCCKeySizeAES256,
        UInt32(msec));
    return actualRoundCount
}
```

Ejemplo de uso:

```
let saltData       = [UInt8]([0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let passwordString = "password"
let delayMsec      = 100

let rounds = pbkdf2SHA1Calibrate(passwordString, salt:saltData, msec:delayMsec)
print("For \(delayMsec) msec delay, rounds: \(rounds)")
```

Ejemplo de salida:

Para 100 ms de retraso, rondas: 94339

Calibración de derivación de clave basada en contraseña (Swift 3)

Determine la cantidad de rondas PRF que se utilizarán para un retraso específico en la plataforma actual.

Varios parámetros están predeterminados para valores representativos que no deberían afectar materialmente el conteo de la ronda.

```
password Sample password.
salt     Sample salt.
msec     Targeted duration we want to achieve for a key derivation.

returns  The number of iterations to use for the desired processing time.


func pbkdf2SHA1Calibrate(password: String, salt: Data, msec: Int) -> UInt32 {
    let actualRoundCount: UInt32 = CCCalibratePBKDF(
        CCPBKDFAlgorithm(kCCPBKDF2),
        password.utf8.count,
        salt.count,
        CCPseudoRandomAlgorithm(kCCPRFHmacAlgSHA1),
        kCCKeySizeAES256,
        UInt32(msec));
    return actualRoundCount
}
```

Ejemplo de uso:

```
let saltData       = Data(bytes: [0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let passwordString = "password"
let delayMsec      = 100

let rounds = pbkdf2SHA1Calibrate(password:passwordString, salt:saltData, msec:delayMsec)
print("For \(delayMsec) msec delay, rounds: \(rounds)")
```

Ejemplo de salida:

```
For 100 msec delay, rounds: 93457
```

Lea Derivación de clave PBKDF2 en línea: https://riptutorial.com/es/swift/topic/7053/derivacion-de-clave-pbkdf2

Capítulo 22: Entrar en Swift

Observaciones

println y debugPrintln eliminaron en Swift 2.0.

Fuentes:

https://developer.apple.com/library/content/technotes/tn2347/_index.html
http://ericasadun.com/2015/05/22/swift-logging/

http://www.dotnetperls.com/print-swift

Examples

Imprimir Debug

Debug Print muestra la representación de instancia más adecuada para la depuración.

```
print("Hello")
debugPrint("Hello")

let dict = ["foo": 1, "bar": 2]

print(dict)
debugPrint(dict)
```

Rendimientos

```
>>> Hello
>>> "Hello"
>>> [foo: 1, bar: 2]
>>> ["foo": 1, "bar": 2]
```

Esta información extra puede ser muy importante, por ejemplo:

```
let wordArray = ["foo", "bar", "food, bars"]

print(wordArray)
debugPrint(wordArray)
```

Rendimientos

```
>>> [foo, bar, food, bars]
>>> ["foo", "bar", "food, bars"]
```

Observe cómo en la primera salida parece que hay 4 elementos en la matriz en lugar de 3. Por razones como esta, es preferible al depurar utilizar debugPrint

Actualización de una clase de depuración e impresión de valores.

```
struct Foo: Printable, DebugPrintable {
    var description: String {return "Clear description of the object"}
```

```
    var debugDescription: String {return "Helpful message for debugging"}
}

var foo = Foo()

print(foo)
debugPrint(foo)

>>> Clear description of the object
>>> Helpful message for debugging
```

tugurio

dump imprime el contenido de un objeto a través de la reflexión (reflejo).

Vista detallada de una matriz:

```
let names = ["Joe", "Jane", "Jim", "Joyce"]
dump(names)
```

Huellas dactilares:

```
▿ 4 elementos
- [0]: Joe
- [1]: Jane
- [2]: Jim
- [3]: Joyce
```

Para un diccionario:

```
let attributes = ["foo": 10, "bar": 33, "baz": 42]
dump(attributes)
```

Huellas dactilares:

```
▿ 3 pares clave / valor
▿ [0]: (2 elementos)
- .0: bar
- .1: 33
▿ [1]: (2 elementos)
- .0: baz
- .1: 42
▿ [2]: (2 elementos)
- .0: foo
- .1: 10
```

dump se declara como dump(_:name:indent:maxDepth:maxItems:) .

El primer parámetro no tiene etiqueta.

Hay otros parámetros disponibles, como name para establecer una etiqueta para el objeto que se está inspeccionando:

```
dump(attributes, name: "mirroring")
```

Huellas dactilares:

```
▿ reflejo: 3 pares clave / valor
▿ [0]: (2 elementos)
- .0: bar
- .1: 33
```

```
▿ [1]: (2 elementos)
- .0: baz
- .1: 42
▿ [2]: (2 elementos)
- .0: foo
- .1: 10
```

También puede optar por imprimir solo un cierto número de elementos con maxItems: para analizar el objeto hasta una cierta profundidad con maxDepth: y para cambiar la sangría de los objetos impresos con indent:

imprimir () vs dump ()

Muchos de nosotros empezamos a depurar con print() simple print() . Digamos que tenemos tal clase:

```
class Abc {
    let a = "aa"
    let b = "bb"
}
```

y tenemos una instancia de Abc como tal:

```
let abc = Abc()
```

Cuando ejecutamos la print() en la variable, la salida es

```
App.Abc
```

mientras dump() salida

```
App.Abc #0
- a: "aa"
- b: "bb"
```

Como se ve, dump() genera toda la jerarquía de clases, mientras que print() simplemente genera el nombre de la clase.

Por lo tanto, dump() es especialmente útil para la depuración de la interfaz de usuario

```
let view = UIView(frame: CGRect(x: 0, y: 0, width: 100, height: 100))
```

Con dump(view) obtenemos:

```
 - <UIView: 0x108a0cde0; frame = (0 0; 100 100); layer = <CALayer: 0x159340cb0>> #0
    - super: UIResponder
      - NSObject
```

Mientras print(view) obtenemos:

```
<UIView: 0x108a0cde0; frame = (0 0; 100 100); layer = <CALayer: 0x159340cb0>>
```

Hay más información sobre la clase con dump() , por lo que es más útil para depurar la propia clase.

imprimir vs NSLog

En swift podemos usar las funciones print() y NSLog() para imprimir algo en la consola Xcode.

Pero hay muchas diferencias en print() funciones print() y NSLog() , tales como:

1 TimeStamp: NSLog() imprimirá la marca de tiempo junto con la cadena que le pasamos, pero print() no imprimirá la marca de tiempo.
p.ej

```
let array = [1, 2, 3, 4, 5]
print(array)
NSLog(array.description)
```

Salida:

```
[1, 2, 3, 4, 5]
2017-05-31 13: 14: 38.582 Nombre del proyecto [2286: 7473287] [1, 2, 3, 4, 5]
```

También imprimirá **ProjectName** junto con la marca de tiempo.

2 Solo cadena: NSLog() solo toma cadena como entrada, pero print() puede imprimir cualquier tipo de entrada que se le pase.
p.ej

```
let array = [1, 2, 3, 4, 5]
print(array) //prints [1, 2, 3, 4, 5]
NSLog(array) //error: Cannot convert value of type [Int] to expected argument type 'String'
```

3 Rendimiento: la función NSLog() es muy **lenta en** comparación con print() función print() .

4 Sincronización: NSLog() maneja el uso simultáneo de un entorno de subprocesos múltiples e imprime la salida sin superponerse. Pero print() no manejará estos casos y confunde mientras realiza la publicación.

5 Consola del dispositivo: NSLog() genera salidas en la consola del dispositivo, podemos ver esta salida conectando nuestro dispositivo a Xcode. print() no imprimirá la salida en la consola del dispositivo.

Lea Entrar en Swift en línea: https://riptutorial.com/es/swift/topic/3966/entrar-en-swift

Capítulo 23: Enums

Observaciones

Al igual que las estructuras y a diferencia de las clases, las enumeraciones son tipos de valor y se copian en lugar de hacer referencia cuando se pasan.

Para obtener más información sobre las enumeraciones, consulte El lenguaje de programación Swift .

Examples

Enumeraciones basicas

Una enumeración proporciona un conjunto de valores relacionados:

```
enum Direction {
    case up
    case down
    case left
    case right
}

enum Direction { case up, down, left, right }
```

Los valores de enumeración se pueden usar por su nombre completo, pero puede omitir el nombre de tipo cuando se puede inferir:

```
let dir = Direction.up
let dir: Direction = Direction.up
let dir: Direction = .up

// func move(dir: Direction)...
move(Direction.up)
move(.up)

obj.dir = Direction.up
obj.dir = .up
```

La forma más fundamental de comparar / extraer valores de enumeración es con una instrucción de switch :

```
switch dir {
case .up:
    // handle the up case
case .down:
    // handle the down case
case .left:
    // handle the left case
case .right:
    // handle the right case
}
```

Las enumeraciones simples son automáticamente Hashable , Equatable y tienen conversiones de cadena:

```
if dir == .down { ... }
```

```
let dirs: Set<Direction> = [.right, .left]

print(Direction.up)  // prints "up"
debugPrint(Direction.up)  // prints "Direction.up"
```

Enums con valores asociados.

Los casos de enumeración pueden contener una o más **cargas útiles** (**valores asociados**):

```
enum Action {
    case jump
    case kick
    case move(distance: Float)  // The "move" case has an associated distance
}
```

La carga útil debe proporcionarse al instanciar el valor de enumeración:

```
performAction(.jump)
performAction(.kick)
performAction(.move(distance: 3.3))
performAction(.move(distance: 0.5))
```

La instrucción switch puede extraer el valor asociado:

```
switch action {
case .jump:
    ...
case .kick:
    ...
case .move(let distance):  // or case let .move(distance):
    print("Moving: \(distance)")
}
```

Una extracción de un solo caso se puede hacer usando el if case :

```
if case .move(let distance) = action {
    print("Moving: \(distance)")
}
```

La sintaxis del guard case se puede utilizar para la extracción posterior del uso:

```
guard case .move(let distance) = action else {
    print("Action is not move")
    return
}
```

Las enumeraciones con valores asociados no son Equatable de forma predeterminada. La implementación del operador == debe hacerse manualmente:

```
extension Action: Equatable { }

func ==(lhs: Action, rhs: Action) -> Bool {
    switch lhs {
    case .jump: if case .jump = rhs { return true }
    case .kick: if case .kick = rhs { return true }
    case .move(let lhsDistance): if case .move (let rhsDistance) = rhs { return lhsDistance ==
rhsDistance }
```

```
    }
    return false
}
```

Cabida indirecta

Normalmente, las enumeraciones no pueden ser recursivas (porque requerirían almacenamiento infinito):

```
enum Tree<T> {
    case leaf(T)
    case branch(Tree<T>, Tree<T>)  // error: recursive enum 'Tree<T>' is not marked 'indirect'
}
```

La palabra clave **indirect** hace que la enumeración almacene su carga útil con una capa de direccionamiento indirecto, en lugar de almacenarla en línea. Puede utilizar esta palabra clave en un solo caso:

```
enum Tree<T> {
    case leaf(T)
    indirect case branch(Tree<T>, Tree<T>)
}

let tree = Tree.branch(.leaf(1), .branch(.leaf(2), .leaf(3)))
```

indirect también funciona en toda la enumeración, haciendo que cualquier caso sea indirecto cuando sea necesario:

```
indirect enum Tree<T> {
    case leaf(T)
    case branch(Tree<T>, Tree<T>)
}
```

Valores crudos y hash

Las enumeraciones sin carga útil pueden tener *valores sin procesar* de cualquier tipo literal:

```
enum Rotation: Int {
    case up = 0
    case left = 90
    case upsideDown = 180
    case right = 270
}
```

Las enumeraciones sin ningún tipo específico no exponen la propiedad rawValue

```
enum Rotation {
    case up
    case right
    case down
    case left
}

let foo = Rotation.up
foo.rawValue //error
```

Se asume que los valores brutos enteros comienzan en 0 y aumentan monótonamente:

```
enum MetasyntacticVariable: Int {
    case foo  // rawValue is automatically 0
    case bar  // rawValue is automatically 1
    case baz = 7
    case quux  // rawValue is automatically 8
}
```

Los valores en bruto de la cadena se pueden sintetizar automáticamente:

```
enum MarsMoon: String {
    case phobos  // rawValue is automatically "phobos"
    case deimos  // rawValue is automatically "deimos"
}
```

Una enumeración en bruto se ajusta automáticamente a RawRepresentable . Puede obtener el valor bruto correspondiente de un valor de enumeración con .rawValue :

```
func rotate(rotation: Rotation) {
    let degrees = rotation.rawValue
    ...
}
```

También puede crear una enumeración a *partir de* un valor en bruto usando init?(rawValue:) :

```
let rotation = Rotation(rawValue: 0)  // returns Rotation.Up
let otherRotation = Rotation(rawValue: 45)  // returns nil (there is no Rotation with rawValue
45)

if let moon = MarsMoon(rawValue: str) {
    print("Mars has a moon named \(str)")
} else {
    print("Mars doesn't have a moon named \(str)")
}
```

Si desea obtener el valor de hash de una enumeración específica, puede acceder a su valor de hash, el valor de hash devolverá el índice de la enumeración comenzando desde cero.

```
let quux = MetasyntacticVariable(rawValue: 8)// rawValue is 8
quux?.hashValue //hashValue is 3
```

Inicializadores

Las enumeraciones pueden tener métodos personalizados de inicio que pueden ser más útiles que el init?(rawValue:) predeterminado init?(rawValue:) . Enums también puede almacenar valores también. Esto puede ser útil para almacenar los valores con los que se inicializaron y recuperar ese valor más adelante.

```
enum CompassDirection {
    case north(Int)
    case south(Int)
    case east(Int)
    case west(Int)

    init?(degrees: Int) {
        switch degrees {
        case 0...45:
            self = .north(degrees)
        case 46...135:
```

```
            self = .east(degrees)
        case 136...225:
            self = .south(degrees)
        case 226...315:
            self = .west(degrees)
        case 316...360:
            self = .north(degrees)
        default:
            return nil
        }
    }

    var value: Int = {
        switch self {
            case north(let degrees):
                return degrees
            case south(let degrees):
                return degrees
            case east(let degrees):
                return degrees
            case west(let degrees):
                return degrees
        }
    }
}
```

Usando ese inicializador podemos hacer esto:

```
var direction = CompassDirection(degrees: 0) // Returns CompassDirection.north
direction = CompassDirection(degrees: 90) // Returns CompassDirection.east
print(direction.value) //prints 90
direction = CompassDirection(degrees: 500) // Returns nil
```

Las enumeraciones comparten muchas características con clases y estructuras

Las enumeraciones en Swift son mucho más poderosas que algunas de sus contrapartes en otros idiomas, como C. Comparten muchas características con clases y estructuras , como la definición de inicializadores , propiedades computadas , métodos de instancia , conformidades de protocolo y extensiones .

```
protocol ChangesDirection {
    mutating func changeDirection()
}

enum Direction {

    // enumeration cases
    case up, down, left, right

    // initialise the enum instance with a case
    // that's in the opposite direction to another
    init(oppositeTo otherDirection: Direction) {
        self = otherDirection.opposite
    }

    // computed property that returns the opposite direction
    var opposite: Direction {
        switch self {
        case .up:
            return .down
```

```
        case .down:
            return .up
        case .left:
            return .right
        case .right:
            return .left
        }
    }
}

// extension to Direction that adds conformance to the ChangesDirection protocol
extension Direction: ChangesDirection {
    mutating func changeDirection() {
        self = .left
    }
}
```

```
var dir = Direction(oppositeTo: .down) // Direction.up

dir.changeDirection() // Direction.left

let opposite = dir.opposite // Direction.right
```

Enumeraciones anidadas

Puede anidar enumeraciones una dentro de otra, esto le permite estructurar enumeraciones jerárquicas para ser más organizadas y claras.

```
enum Orchestra {
    enum Strings {
        case violin
        case viola
        case cello
        case doubleBasse
    }

    enum Keyboards {
        case piano
        case celesta
        case harp
    }

    enum Woodwinds {
        case flute
        case oboe
        case clarinet
        case bassoon
        case contrabassoon
    }
}
```

Y puedes usarlo así:

```
let instrment1 = Orchestra.Strings.viola
let instrment2 = Orchestra.Keyboards.piano
```

Lea Enums en línea: https://riptutorial.com/es/swift/topic/224/enums

Capítulo 24: Estructuras

Examples

Fundamentos de Estructuras

```
struct Repository {
    let identifier: Int
    let name: String
    var description: String?
}
```

Esto define una estructura de Repository con tres propiedades almacenadas, un identifier entero, un name cadena y una description cadena opcional. El identifier y el name son constantes, ya que se han declarado usando la palabra clave let . Una vez configurados durante la inicialización, no se pueden modificar. La descripción es una variable. Al modificarlo se actualiza el valor de la estructura.

Los tipos de estructura reciben automáticamente un inicializador de memberwise si no definen ninguno de sus propios inicializadores personalizados. La estructura recibe un inicializador de memberwise incluso si tiene propiedades almacenadas que no tienen valores predeterminados.

Repository contiene tres propiedades almacenadas de las cuales solo la description tiene un valor predeterminado (nil). Además, no define inicializadores propios, por lo que recibe un inicializador de memberwise de forma gratuita:

```
let newRepository = Repository(identifier: 0, name: "New Repository", description: "Brand New
Repository")
```

Las estructuras son tipos de valor

A diferencia de las clases, que se pasan por referencia, las estructuras se pasan a través de la copia:

```
first = "Hello"
second = first
first += " World!"
// first == "Hello World!"
// second == "Hello"
```

La cadena es una estructura, por lo tanto, se copia en la asignación.

Las estructuras tampoco se pueden comparar usando el operador de identidad:

```
window0 === window1 // works because a window is a class instance
"hello" === "hello" // error: binary operator '===' cannot be applied to two 'String' operands
```

Cada dos instancias de estructura se consideran idénticas si se comparan iguales.

En conjunto, estos rasgos que diferencian las estructuras de las clases son lo que hace que las estructuras valoren los tipos.

Mutando un Struct

Un método de una estructura que cambie el valor de la estructura en sí debe tener un prefijo con la palabra clave que mutating

```
struct Counter {
    private var value = 0

    mutating func next() {
        value += 1
    }
}
```

Cuando puedes usar métodos de mutación

Los métodos de mutating solo están disponibles en valores de estructura dentro de variables.

```
var counter = Counter()
counter.next()
```

Cuando NO puedes usar métodos de mutación

Por otro lado, los métodos de mutating NO están disponibles en valores de estructura dentro de constantes

```
let counter = Counter()
counter.next()
//  error: cannot use mutating member on immutable value: 'counter' is a 'let' constant
```

Las estructuras no pueden heredar

A diferencia de las clases, las estructuras no pueden heredar:

```
class MyView: NSView { }  // works

struct MyInt: Int { } // error: inheritance from non-protocol type 'Int'
```

Las estructuras, sin embargo, pueden adoptar protocolos:

```
struct Vector: Hashable { ... }  // works
```

Accediendo miembros de struct

En Swift, las estructuras usan una simple "sintaxis de puntos" para acceder a sus miembros.

Por ejemplo:

```
struct DeliveryRange {
  var range: Double
  let center: Location
}
let storeLocation = Location(latitude: 44.9871,
                             longitude: -93.2758)
var pizzaRange = DeliveryRange(range: 200,
                               center: storeLocation)
```

Puede acceder (imprimir) el rango de esta manera:

```
    print(pizzaRange.range) // 200
```

Incluso puede acceder a miembros de miembros usando la sintaxis de puntos:

```
print(pizzaRange.center.latitude) // 44.9871
```

De manera similar a cómo puedes leer valores con sintaxis de puntos, también puedes asignarlos.

```
pizzaRange.range = 250
```

Lea Estructuras en línea: https://riptutorial.com/es/swift/topic/255/estructuras

Capítulo 25: Extensiones

Observaciones

Puedes leer más sobre las extensiones en The Swift Programming Language .

Examples

Variables y funciones

Las extensiones pueden contener funciones y variables de obtención calculadas / constantes. Están en el formato.

```
extension ExtensionOf {
    //new functions and get-variables
}
```

Para hacer referencia a la instancia del objeto extendido, self puede usar self , tal como podría usarse

Para crear una extensión de String que agregue una función .length() que devuelva la longitud de la cadena, por ejemplo

```
extension String {
    func length() -> Int {
        return self.characters.count
    }
}

"Hello, World!".length() // 13
```

Las extensiones también pueden contener variables get . Por ejemplo, agregar una variable .length a la cadena que devuelve la longitud de la cadena

```
extension String {
    var length: Int {
        get {
            return self.characters.count
        }
    }
}

"Hello, World!".length // 13
```

Inicializadores en extensiones

Las extensiones pueden contener inicializadores de conveniencia. Por ejemplo, un inicializador failable para Int que acepta un NSString :

```
extension Int {
    init?(_ string: NSString) {
        self.init(string as String)  // delegate to the existing Int.init(String) initializer
    }
}

let str1: NSString = "42"
Int(str1) // 42
```

```
let str2: NSString = "abc"
Int(str2) // nil
```

¿Qué son las extensiones?

Las extensiones se utilizan para ampliar la funcionalidad de los tipos existentes en Swift. Las extensiones pueden agregar subíndices, funciones, inicializadores y propiedades computadas. También pueden hacer que los tipos se ajusten a los protocolos .

Supongamos que desea poder calcular el factorial de un Int . Puedes agregar una propiedad computada en una extensión:

```
extension Int {
    var factorial: Int {
        return (1..<self+1).reduce(1, combine: *)
    }
}
```

Luego puede acceder a la propiedad como si se hubiera incluido en la API Int original.

```
let val1: Int = 10

val1.factorial  // returns 3628800
```

Extensiones de protocolo

Una característica muy útil de Swift 2.2 es tener la capacidad de extender protocolos.

Funciona casi como clases abstractas cuando se trata de una funcionalidad que desea que esté disponible en todas las clases que implementan algún protocolo (sin tener que heredar de una clase común básica).

```
protocol FooProtocol {
    func doSomething()
}

extension FooProtocol {
    func doSomething() {
        print("Hi")
    }
}

class Foo: FooProtocol {
    func myMethod() {
        doSomething() // By just implementing the protocol this method is available
    }
}
```

Esto también es posible usando genéricos.

Restricciones

Es posible escribir un método en un tipo genérico que sea más restrictivo usando la oración where.

```
extension Array where Element: StringLiteralConvertible {
```

```
    func toUpperCase() -> [String] {
        var result = [String]()
        for value in self {
            result.append(String(value).uppercaseString)
        }
        return result
    }
}
```

Ejemplo de uso

```
let array = ["a","b","c"]
let resultado = array.toUpperCase()
```

Qué son las extensiones y cuándo usarlas.

Las extensiones agregan nueva funcionalidad a una clase, estructura, enumeración o tipo de protocolo existente. Esto incluye la capacidad de extender tipos para los que no tiene acceso al código fuente original.

Extensiones en Swift pueden:

- Añadir propiedades computadas y propiedades de tipo computadas
- Definir métodos de instancia y métodos de tipo.
- Proporcionar nuevos inicializadores
- Definir subíndices
- Definir y utilizar nuevos tipos anidados.
- Hacer que un tipo existente se ajuste a un protocolo.

Cuándo usar Swift Extensions:

- Funcionalidad adicional para Swift
- Funcionalidad adicional a UIKit / Foundation
- Funcionalidad adicional sin alterar el código de otras personas.
- Clases de desglose en: Datos / Funcionalidad / Delegado

Cuando no usar:

- Extiende tus propias clases desde otro archivo

Ejemplo simple:

```
extension Bool {
    public mutating func toggle() -> Bool {
        self = !self
        return self
    }
}

var myBool: Bool = true
print(myBool.toggle()) // false
```

Fuente

Subíndices

Las extensiones pueden agregar nuevos subíndices a un tipo existente.

Este ejemplo obtiene el carácter dentro de una cadena usando el índice dado:

2.2

```
extension String {
    subscript(index: Int) -> Character {
        let newIndex = startIndex.advancedBy(index)
        return self[newIndex]
    }
}

var myString = "StackOverFlow"
print(myString[2]) // a
print(myString[3]) // c
```

3.0

```
extension String {
    subscript(offset: Int) -> Character {
        let newIndex = self.index(self.startIndex, offsetBy: offset)
        return self[newIndex]
    }
}

var myString = "StackOverFlow"
print(myString[2]) // a
print(myString[3]) // c
```

Lea Extensiones en línea: https://riptutorial.com/es/swift/topic/324/extensiones

Capítulo 26: Funcionar como ciudadanos de primera clase en Swift

Introducción

Funciones como miembros de primera clase significa que puede disfrutar de privilegios al igual que los objetos. Puede asignarse a una variable, pasarse a una función como parámetro o puede usarse como tipo de retorno.

Examples

Asignando función a una variable

```
struct Mathematics
{
    internal func performOperation(inputArray: [Int], operation: (Int)-> Int)-> [Int]
    {
        var processedArray = [Int]()

        for item in inputArray
        {
            processedArray.append(operation(item))
        }

        return processedArray
    }


    internal func performComplexOperation(valueOne: Int)-> ((Int)-> Int)
    {
        return
            ({
                 return valueOne + $0
            })
    }

}


let arrayToBeProcessed = [1,3,5,7,9,11,8,6,4,2,100]

let math = Mathematics()

func add2(item: Int)-> Int
{
    return (item + 2)
}

// assigning the function to a variable and then passing it to a function as param
let add2ToMe = add2
print(math.performOperation(inputArray: arrayToBeProcessed, operation: add2ToMe))
```

Salida:

```
[3, 5, 7, 9, 11, 13, 10, 8, 6, 4, 102]
```

Del mismo modo lo anterior podría lograrse mediante un `closure`

```
// assigning the closure to a variable and then passing it to a function as param
```

```
let add2 = {(item: Int)-> Int in return item + 2}
print(math.performOperation(inputArray: arrayToBeProcessed, operation: add2))
```

Pasar la función como un argumento a otra función, creando así una función de orden superior

```
func multiply2(item: Int)-> Int
{
    return (item + 2)
}


let multiply2ToMe = multiply2

// passing the function directly to the function as param
print(math.performOperation(inputArray: arrayToBeProcessed, operation: multiply2ToMe))
```

Salida:

```
[3, 5, 7, 9, 11, 13, 10, 8, 6, 4, 102]
```

Del mismo modo lo anterior podría lograrse mediante un `closure`

```
// passing the closure directly to the function as param
print(math.performOperation(inputArray: arrayToBeProcessed, operation: { $0 * 2 }))
```

Función como tipo de retorno de otra función.

```
// function as return type
print(math.performComplexOperation(valueOne: 4)(5))
```

Salida:

9

Lea Funcionar como ciudadanos de primera clase en Swift en línea:
https://riptutorial.com/es/swift/topic/8618/funcionar-como-ciudadanos-de-primera-clase-en-swift

Capítulo 27: Funciones

Examples

Uso básico

Las funciones pueden ser declaradas sin parámetros o un valor de retorno. La única información requerida es un nombre (hello en este caso).

```
func hello()
{
    print("Hello World")
}
```

Llame a una función sin parámetros escribiendo su nombre seguido de un par de paréntesis vacío.

```
hello()
//output: "Hello World"
```

Funciones con parámetros

Las funciones pueden tomar parámetros para que su funcionalidad pueda ser modificada. Los parámetros se dan como una lista separada por comas con sus tipos y nombres definidos.

```
func magicNumber(number1: Int)
{
    print("\(number1) Is the magic number")
}
```

Nota: La sintaxis \(number1) es una Interpolación de cadena básica y se utiliza para insertar el número entero en la Cadena.

Las funciones con parámetros se llaman especificando la función por nombre y proporcionando un valor de entrada del tipo utilizado en la declaración de función.

```
magicNumber(5)
//output: "5 Is the magic number
let example: Int = 10
magicNumber(example)
//output: "10 Is the magic number"
```

Cualquier valor de tipo Int podría haber sido usado.

```
func magicNumber(number1: Int, number2: Int)
{
    print("\(number1 + number2) Is the magic number")
}
```

Cuando una función usa múltiples parámetros, el nombre del primer parámetro no es necesario para el primero, pero está en los parámetros posteriores.

```
let ten: Int = 10
let five: Int = 5
magicNumber(ten,number2: five)
//output: "15 Is the magic number"
```

Utilice nombres de parámetros externos para hacer que las llamadas de función sean más legibles.

```
func magicNumber(one number1: Int, two number2: Int)
{
    print("\(number1 + number2) Is the magic number")
}

let ten: Int = 10
let five: Int = 5
magicNumber(one: ten, two: five)
```

Establecer el valor predeterminado en la declaración de función le permite llamar a la función sin dar valores de entrada.

```
func magicNumber(one number1: Int = 5, two number2: Int = 10)
{
    print("\(number1 + number2) Is the magic number")
}

magicNumber()
//output: "15 Is the magic number"
```

Valores de retorno

Las funciones pueden devolver valores especificando el tipo después de la lista de parámetros.

```
func findHypotenuse(a: Double, b: Double) -> Double
{
    return sqrt((a * a) + (b * b))
}

let c = findHypotenuse(3, b: 5)
//c = 5.830951894845301
```

Las funciones también pueden devolver múltiples valores utilizando tuplas.

```
func maths(number: Int) -> (times2: Int, times3: Int)
{
    let two = number * 2
    let three = number * 3
    return (two, three)
}
let resultTuple = maths(5)
//resultTuple = (10, 15)
```

Errores de lanzamiento

Si desea que una función pueda generar errores, debe agregar la palabra clave de throws después de los paréntesis que contienen los argumentos:

```
func errorThrower()throws -> String {}
```

Cuando quieras lanzar un error, usa la palabra clave throw :

```
func errorThrower()throws -> String {
  if true {
    return "True"
  } else {
    // Throwing an error
```

```
    throw Error.error
  }
}
```

Si desea llamar a una función que puede generar un error, debe usar la palabra clave try en un bloque do :

```
do {
  try errorThrower()
}
```

Para más información sobre los errores de Swift: Errores

Métodos

Los métodos de instancia son funciones que pertenecen a instancias de un tipo en Swift (una clase , estructura , enumeración o protocolo). **Los métodos de tipo** se llaman en un tipo en sí.

Métodos de instancia

Los métodos de instancia se definen con una declaración de func dentro de la definición del tipo, o en una extensión .

```
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
}
```

El método de instancia increment() se llama en una instancia de la clase Counter :

```
let counter = Counter()  // create an instance of Counter class
counter.increment()      // call the instance method on this instance
```

Métodos de tipo

Los métodos de tipo se definen con las palabras clave static func . (Para las clases, class func define un método de tipo que puede ser reemplazado por subclases).

```
class SomeClass {
    class func someTypeMethod() {
        // type method implementation goes here
    }
}
```

```
SomeClass.someTypeMethod()  // type method is called on the SomeClass type itself
```

Parámetros de entrada

Las funciones pueden modificar los parámetros pasados a ellos si están marcados con la palabra clave inout . Al pasar un inout parámetro a una función, la persona que llama debe agregar una & al que se pasa la variable.

```
func updateFruit(fruit: inout Int) {
    fruit -= 1
```

```
}

var apples = 30 // Prints "There's 30 apples"
print("There's \(apples) apples")

updateFruit(fruit: &apples)

print("There's now \(apples) apples") // Prints "There's 29 apples".
```

Esto permite que la semántica de referencia se aplique a tipos que normalmente tendrían una semántica de valor.

Sintaxis de cierre de seguimiento

Cuando el último parámetro de una función es un cierre.

```
func loadData(id: String, completion:(result: String) -> ()) {
    // ...
    completion(result:"This is the result data")
}
```

la función se puede invocar mediante la sintaxis de cierre de seguimiento

```
loadData("123") { result in
    print(result)
}
```

Los operadores son funciones

Operadores como + , - , ?? son un tipo de función nombrada usando símbolos en lugar de letras. Se invocan de forma diferente a las funciones:

- Prefijo: **-** x
- Infijo: x **+** y
- Postfijo: x **++**

Puede leer más sobre operadores básicos y operadores avanzados en The Swift Programming Language.

Parámetros variables

A veces, no es posible enumerar la cantidad de parámetros que una función podría necesitar. Considere una función de sum :

```
func sum(_ a: Int, _ b: Int) -> Int {
    return a + b
}
```

Esto funciona bien para encontrar la suma de dos números, pero para encontrar la suma de tres tendríamos que escribir otra función:

```
func sum(_ a: Int, _ b: Int, _ c: Int) -> Int {
    return a + b + c
}
```

y uno con cuatro parámetros necesitaría otro, y así sucesivamente. Swift hace posible definir una función con un número variable de parámetros utilizando una secuencia de tres períodos: ... Por ejemplo,

```
func sum(_ numbers: Int...) -> Int {
    return numbers.reduce(0, combine: +)
}
```

Observe cómo el parámetro numbers , que es variadic, se fusiona en una única Array de tipo [Int] . Esto es cierto en general, los parámetros variables del tipo T... son accesibles como [T] .

Esta función ahora se puede llamar así:

```
let a = sum(1, 2) // a == 3
let b = sum(3, 4, 5, 6, 7) // b == 25
```

Un parámetro variadic en Swift no tiene que aparecer al final de la lista de parámetros, pero solo puede haber uno en cada firma de función.

A veces, es conveniente poner un tamaño mínimo en el número de parámetros. Por ejemplo, realmente no tiene sentido tomar la sum de ningún valor. Una manera fácil de hacer cumplir esto es colocando algunos parámetros requeridos no variados y luego agregando el parámetro variadic después. Para asegurarnos de que solo se pueda llamar a la sum con al menos dos parámetros, podemos escribir

```
func sum(_ n1: Int, _ n2: Int, _ numbers: Int...) -> Int {
    return numbers.reduce(n1 + n2, combine: +)
}

sum(1, 2) // ok
sum(3, 4, 5, 6, 7) // ok
sum(1) // not ok
sum() // not ok
```

Subíndices

Las clases, estructuras y enumeraciones pueden definir subíndices, que son accesos directos para acceder a los elementos miembros de una colección, lista o secuencia.

Ejemplo

```
struct DaysOfWeek {

  var days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]

  subscript(index: Int) -> String {
    get {
      return days[index]
    }
    set {
      days[index] = newValue
    }
  }
}
```

Uso de subíndices

```
var week = DaysOfWeek()
//you access an element of an array at index by array[index].
debugPrint(week[1])
debugPrint(week[0])
week[0] = "Sunday"
debugPrint(week[0])
```

Opciones de subíndices:

Los subíndices pueden tomar cualquier número de parámetros de entrada, y estos parámetros de entrada pueden ser de cualquier tipo. Los subíndices también pueden devolver cualquier tipo. Los subíndices pueden usar parámetros variables y parámetros variadic, pero no pueden usar parámetros in-out o proporcionar valores de parámetros predeterminados.

Ejemplo:

```
struct Food {

  enum MealTime {
    case Breakfast, Lunch, Dinner
  }

  var meals: [MealTime: String] = [:]

  subscript (type: MealTime) -> String? {
    get {
      return meals[type]
    }
    set {
      meals[type] = newValue
    }
  }
}
```

Uso

```
var diet = Food()
diet[.Breakfast] = "Scrambled Eggs"
diet[.Lunch] = "Rice"

debugPrint("I had \(diet[.Breakfast]) for breakfast")
```

Funciones con cierres

El uso de funciones que toman y ejecutan cierres puede ser extremadamente útil para enviar un bloque de código para que se ejecute en otro lugar. Podemos comenzar permitiendo que nuestra función tome un cierre opcional que (en este caso) devolverá Void .

```
func closedFunc(block: (()->Void)? = nil) {
    print("Just beginning")

    if let block = block {
        block()
    }
}
```

Ahora que nuestra función ha sido definida, llamémosla y pasemos algún código:

```
closedFunc() { Void in
    print("Over already")
}
```

Al utilizar un **cierre final** con nuestra llamada de función, podemos pasar el código (en este caso, print) para que se ejecute en algún punto dentro de nuestra función closedFunc() .

El registro debe imprimir:

Recién empezando

Sobre ya

Un caso de uso más específico de esto podría incluir la ejecución de código entre dos clases:

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        let _  = A.init(){Void in self.action(2)}
    }

    func action(i: Int) {
        print(i)
    }
}


class A: NSObject {
    var closure : ()?

    init(closure: (()->Void)? = nil) {
        // Notice how this is executed before the  closure
        print("1")
        // Make sure closure isn't nil
        self.closure = closure?()
    }
}
```

El registro debe imprimir:

1

2

Funciones de paso y retorno.

La siguiente función está devolviendo otra función como su resultado, que puede ser asignada posteriormente a una variable y llamada:

```
func jediTrainer () -> ((String, Int) -> String) {
  func train(name: String, times: Int) -> (String) {
    return "\(name) has been trained in the Force \(times) times"
  }
  return train
}

let train = jediTrainer()
train("Obi Wan", 3)
```

Tipos de funciones

Cada función tiene su propio tipo de función, compuesto por los tipos de parámetros y el tipo de retorno de la función en sí. Por ejemplo la siguiente función:

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

tiene un tipo de función de:

```
(Int, Int) -> (Int)
```

Por lo tanto, los tipos de función se pueden usar como tipos de parámetros o como tipos de retorno para funciones de anidamiento.

Lea Funciones en línea: https://riptutorial.com/es/swift/topic/432/funciones

Capítulo 28: Funciones Swift Advance

Introducción

Las funciones flatMap como map , flatMap , filter , y reduce se utilizan para operar en varios tipos de colecciones como Array y Dictionary. Las funciones avanzadas normalmente requieren poco código y se pueden encadenar para construir una lógica compleja de manera concisa.

Examples

Introducción con funciones avanzadas.

Tomemos un escenario para entender mejor la función avanzada,

```
struct User {
    var name: String
    var age: Int
    var country: String?
}

//User's information
let user1 = User(name: "John", age: 24, country: "USA")
let user2 = User(name: "Chan", age: 20, country: nil)
let user3 = User(name: "Morgan", age: 30, country: nil)
let user4 = User(name: "Rachel", age: 20, country: "UK")
let user5 = User(name: "Katie", age: 23, country: "USA")
let user6 = User(name: "David", age: 35, country: "USA")
let user7 = User(name: "Bob",age: 22, country: nil)

//User's array list
let arrUser = [user1, user2, user3, user4, user5, user6, user7]
```

Función de mapa:

Utilice el mapa para recorrer una colección y aplicar la misma operación a cada elemento de la colección. La función de mapa devuelve una matriz que contiene los resultados de aplicar una función de mapeo o transformación a cada elemento.

```
//Fetch all the user's name from array
let arrUserName = arrUser.map({ $0.name }) // ["John", "Chan", "Morgan", "Rachel", "Katie",
"David", "Bob"]
```

Función de mapa plano:

El uso más simple es como el nombre sugiere para aplanar una colección de colecciones.

```
// Fetch all user country name & ignore nil value.
let arrCountry = arrUser.flatMap({ $0.country }) // ["USA", "UK", "USA", "USA"]
```

Función de filtro:

Use el filtro para recorrer una colección y devuelva una matriz que contenga solo los elementos que coincidan con una condición de inclusión.

```
// Filtering USA user from the array user list.
let arrUSAUsers = arrUser.filter({ $0.country == "USA" }) // [user1, user5, user6]

// User chaining methods to fetch user's name who live in USA
let arrUserList = arrUser.filter({ $0.country == "USA" }).map({ $0.name }) // ["John",
```

```
"Katie", "David"]
```

Reducir:

Use Reducir para combinar todos los elementos de una colección para crear un nuevo valor único.

Swift 2.3: -

```
//Fetch user's total age.
let arrUserAge = arrUser.map({ $0.age }).reduce(0, combine: { $0 + $1 }) //174

//Prepare all user name string with seperated by comma
let strUserName = arrUserName.reduce("", combine: { $0 == "" ? $1 : $0 + ", " + $1 }) // John,
Chan, Morgan, Rachel, Katie, David, Bob
```

Swift 3: -

```
//Fetch user's total age.
let arrUserAge = arrUser.map({ $0.age }).reduce(0, { $0 + $1 }) //174

//Prepare all user name string with seperated by comma
let strUserName = arrUserName.reduce("", { $0 == "" ? $1 : $0 + ", " + $1 }) // John, Chan,
Morgan, Rachel, Katie, David, Bob
```

Aplanar matriz multidimensional

Para aplanar la matriz multidimensional en una sola dimensión, se utilizan las funciones de avance de flatMap. Otro caso de uso es ignorar el valor nulo de los valores de matriz y asignación. Vamos a ver con el ejemplo:

Supongamos que tenemos una matriz multidimensional de ciudades y queremos clasificar la lista de nombres de ciudades en orden ascendente. En ese caso podemos usar la función flatMap como:

```
let arrStateName = [["Alaska", "Iowa", "Missouri", "New Mexico"], ["New York", "Texas",
"Washington", "Maryland"], ["New Jersey", "Virginia", "Florida", "Colorado"]]
```

Preparando una lista unidimensional a partir de una matriz multidimensional,

```
let arrFlatStateList = arrStateName.flatMap({ $0 }) // ["Alaska", "Iowa", "Missouri", "New
Mexico", "New York", "Texas", "Washington", "Maryland", "New Jersey", "Virginia", "Florida",
"Colorado"]
```

Para ordenar valores de matriz, podemos usar la operación de encadenamiento o ordenar la matriz plana. Aquí abajo el ejemplo que muestra la operación de encadenamiento,

```
// Swift 2.3 syntax
let arrSortedStateList = arrStateName.flatMap({ $0 }).sort(<) // ["Alaska",     "Colorado",
"Florida", "Iowa", "Maryland", "Missouri", "New Jersey", "New Mexico", "New York", "Texas",
"Virginia", "Washington"]

// Swift 3 syntax
let arrSortedStateList = arrStateName.flatMap({ $0 }).sorted(by: <) // ["Alaska", "Colorado",
"Florida", "Iowa", "Maryland", "Missouri", "New Jersey", "New Mexico", "New York", "Texas",
"Virginia", "Washington"]
```

Lea Funciones Swift Advance en línea: https://riptutorial.com/es/swift/topic/9279/funciones-swift-advance

Capítulo 29: Generar UIImage de Iniciales desde String

Introducción

Esta es una clase que generará un UIImage de las iniciales de una persona. Harry Potter generaría una imagen de HP.

Examples

InicialesImageFactory

```
class InitialsImageFactory: NSObject {

class func imageWith(name: String?) -> UIImage? {

let frame = CGRect(x: 0, y: 0, width: 50, height: 50)
let nameLabel = UILabel(frame: frame)
nameLabel.textAlignment = .center
nameLabel.backgroundColor = .lightGray
nameLabel.textColor = .white
nameLabel.font = UIFont.boldSystemFont(ofSize: 20)
var initials = ""

if let initialsArray = name?.components(separatedBy: " ") {

  if let firstWord = initialsArray.first {
    if let firstLetter = firstWord.characters.first {
      initials += String(firstLetter).capitalized
    }

  }
  if initialsArray.count > 1, let lastWord = initialsArray.last {
    if let lastLetter = lastWord.characters.first {
      initials += String(lastLetter).capitalized
    }

  }
} else {
  return nil
}

nameLabel.text = initials
UIGraphicsBeginImageContext(frame.size)
if let currentContext = UIGraphicsGetCurrentContext() {
  nameLabel.layer.render(in: currentContext)
  let nameImage = UIGraphicsGetImageFromCurrentImageContext()
  return nameImage
}
return nil
}

}
```

Lea Generar UIImage de Iniciales desde String en línea:
https://riptutorial.com/es/swift/topic/10915/generar-uiimage-de-iniciales-desde-string

Capítulo 30: Genéricos

Observaciones

> El código genérico le permite escribir funciones y tipos flexibles y reutilizables que pueden funcionar con cualquier tipo, sujeto a los requisitos que usted defina. Puede escribir código que evite la duplicación y exprese su intención de una manera clara y abstracta.
>
> Los genéricos son una de las características más poderosas de Swift, y gran parte de la biblioteca estándar de Swift está construida con código genérico. Por ejemplo, los tipos Array y Dictionary Swift son colecciones genéricas. Puede crear una matriz que contenga valores de Int , o una matriz que contenga valores de String , o incluso una matriz para cualquier otro tipo que pueda crearse en Swift. De manera similar, puede crear un diccionario para almacenar valores de cualquier tipo específico, y no hay limitaciones sobre lo que ese tipo puede ser.
>
> **Fuente:** lenguaje de programación Swift de Apple

Examples

Restricción de tipos genéricos de marcadores de posición

Es posible forzar los parámetros de tipo de una clase genérica para implementar un protocolo , por ejemplo, Equatable

```
class MyGenericClass<Type: Equatable>{

    var value: Type
    init(value: Type){
        self.value = value
    }

    func getValue() -> Type{
        return self.value
    }

    func valueEquals(anotherValue: Type) -> Bool{
        return self.value == anotherValue
    }
}
```

Cada vez que creamos una nueva MyGenericClass , el parámetro type debe implementar el protocolo Equatable (asegurando que el parámetro type pueda compararse con otra variable del mismo tipo usando ==)

```
let myFloatGeneric = MyGenericClass<Double>(value: 2.71828) // valid
let myStringGeneric = MyGenericClass<String>(value: "My String") // valid

// "Type [Int] does not conform to protocol 'Equatable'"
let myInvalidGeneric = MyGenericClass<[Int]>(value: [2])

let myIntGeneric = MyGenericClass<Int>(value: 72)
print(myIntGeneric.valueEquals(72)) // true
print(myIntGeneric.valueEquals(-274)) // false

// "Cannot convert value of type 'String' to expected argument type 'Int'"
print(myIntGeneric.valueEquals("My String"))
```

Los fundamentos de los genéricos

Los genéricos son marcadores de posición para los tipos, lo que le permite escribir código flexible que se puede aplicar a varios tipos. La ventaja de usar los genéricos en lugar de Any es que aún permiten que el compilador imponga una seguridad de tipo sólida.

Un marcador de posición genérico se define entre paréntesis angulares <> .

Funciones genéricas

Para las funciones , este marcador de posición se coloca después del nombre de la función:

```
/// Picks one of the inputs at random, and returns it
func pickRandom<T>(_ a:T, _ b:T) -> T {
    return arc4random_uniform(2) == 0 ? a : b
}
```

En este caso, el marcador de posición genérico es T Cuando vienes a llamar a la función, Swift puede inferir el tipo de T para ti (ya que simplemente actúa como un marcador de posición para un tipo real).

```
let randomOutput = pickRandom(5, 7) // returns an Int (that's either 5 or 7)
```

Aquí estamos pasando dos enteros a la función. Por lo tanto, Swift está infiriendo T == Int : por lo tanto, la firma de función se infiere que es (Int, Int) -> Int .

Debido al tipo fuerte de seguridad que ofrecen los genéricos, tanto los argumentos como el retorno de la función deben ser del *mismo* tipo. Por lo tanto lo siguiente no se compilará:

```
struct Foo {}

let foo = Foo()

let randomOutput = pickRandom(foo, 5) // error: cannot convert value of type 'Int' to expected
argument type 'Foo'
```

Tipos genéricos

Para utilizar los genéricos con clases , estructuras o enumeraciones , puede definir el marcador de posición genérico después del nombre de tipo.

```
class Bar<T> {
    var baz : T

    init(baz:T) {
        self.baz = baz
    }
}
```

Este marcador de posición genérico requerirá un tipo cuando venga a usar la Bar clase. En este caso, se puede inferir del inicializador init(baz:T) .

```
let bar = Bar(baz: "a string") // bar's type is Bar<String>
```

Aquí se infiere que el marcador de posición T genérico es de tipo String , creando así una instancia de Bar<String> . También puede especificar el tipo explícitamente:

```
let bar = Bar<String>(baz: "a string")
```

Cuando se usa con un tipo, el marcador de posición genérico dado mantendrá su tipo durante toda la vida útil de la instancia dada, y no podrá cambiarse después de la inicialización. Por lo tanto, cuando accede a la propiedad baz , siempre será de tipo String para esta instancia dada.

```
let str = bar.baz // of type String
```

Pasando por tipos genéricos

Cuando llega a pasar por tipos genéricos, en la mayoría de los casos debe ser explícito sobre el tipo de marcador de posición genérico que espera. Por ejemplo, como una entrada de función:

```
func takeABarInt(bar:Bar<Int>) {
    ...
}
```

Esta función solo aceptará una Bar<Int> . Intentar pasar en una instancia de Bar donde el tipo de marcador de posición genérico no es Int dará como resultado un error del compilador.

Nombre genérico de marcador de posición

Los nombres genéricos de los marcadores de posición no se limitan a letras individuales. Si un marcador de posición dado representa un concepto significativo, debe darle un nombre descriptivo. Por ejemplo, Swift's Array tiene un marcador de posición genérico llamado Element , que define el tipo de elemento de una instancia de Array dada.

```
public struct Array<Element> : RandomAccessCollection, MutableCollection {
    ...
}
```

Ejemplos genéricos de clase

Una clase genérica con el parámetro tipo Type

```
class MyGenericClass<Type>{

    var value: Type
    init(value: Type){
        self.value = value
    }

    func getValue() -> Type{
        return self.value
    }

    func setValue(value: Type){
        self.value = value
    }
}
```

Ahora podemos crear nuevos objetos usando un parámetro de tipo

```
let myStringGeneric = MyGenericClass<String>(value: "My String Value")
let myIntGeneric = MyGenericClass<Int>(value: 42)

print(myStringGeneric.getValue()) // "My String Value"
print(myIntGeneric.getValue()) // 42

myStringGeneric.setValue("Another String")
myIntGeneric.setValue(1024)
```

```
print(myStringGeneric.getValue()) // "Another String"
print(myIntGeneric.getValue()) // 1024
```

Los genéricos también se pueden crear con múltiples parámetros de tipo

```
class AnotherGenericClass<TypeOne, TypeTwo, TypeThree>{

    var value1: TypeOne
    var value2: TypeTwo
    var value3: TypeThree
    init(value1: TypeOne, value2: TypeTwo, value3: TypeThree){
        self.value1 = value1
        self.value2 = value2
        self.value3 = value3
    }

    func getValueOne() -> TypeOne{return self.value1}
    func getValueTwo() -> TypeTwo{return self.value2}
    func getValueThree() -> TypeThree{return self.value3}
}
```

Y usado de la misma manera.

```
let myGeneric = AnotherGenericClass<String, Int, Double>(value1: "Value of pi", value2: 3,
value3: 3.14159)

print(myGeneric.getValueOne() is String) // true
print(myGeneric.getValueTwo() is Int) // true
print(myGeneric.getValueThree() is Double) // true
print(myGeneric.getValueTwo() is String) // false

print(myGeneric.getValueOne()) // "Value of pi"
print(myGeneric.getValueTwo()) // 3
print(myGeneric.getValueThree()) // 3.14159
```

Herencia genérica de clase

Las clases genéricas pueden ser heredadas:

```
// Models
class MyFirstModel {
}

class MySecondModel: MyFirstModel {
}

// Generic classes
class MyFirstGenericClass<T: MyFirstModel> {

    func doSomethingWithModel(model: T) {
        // Do something here
    }

}

class MySecondGenericClass<T: MySecondModel>: MyFirstGenericClass<T> {

    override func doSomethingWithModel(model: T) {
```

```
        super.doSomethingWithModel(model)

        // Do more things here
    }

}
```

Usando Genéricos para Simplificar las Funciones de Arreglos

Una función que amplía la funcionalidad de la matriz creando una función de eliminación orientada a objetos.

```
// Need to restrict the extension to elements that can be compared.
// The `Element` is the generics name defined by Array for its item types.
// This restriction also gives us access to `index(of:_)` which is also
// defined in an Array extension with `where Element: Equatable`.
public extension Array where Element: Equatable {
    /// Removes the given object from the array.
    mutating func remove(_ element: Element) {
        if let index = self.index(of: element ) {
            self.remove(at: index)
        } else {
            fatalError("Removal error, no such element:\"\(element)\" in array.\n")
        }
    }
}
```

Uso

```
var myArray = [1,2,3]
print(myArray)

// Prints [1,2,3]
```

Utilice la función para eliminar un elemento sin necesidad de un índice. Sólo pasa el objeto para eliminar.

```
myArray.remove(2)
print(myArray)

// Prints [1,3]
```

Use genéricos para mejorar la seguridad de tipo

Tomemos este ejemplo sin usar genéricos.

```
protocol JSONDecodable {
    static func from(_ json: [String: Any]) -> Any?
}
```

La declaración de protocolo parece estar bien a menos que realmente la uses.

```
let myTestObject = TestObject.from(myJson) as? TestObject
```

¿Por qué tienes que lanzar el resultado a TestObject ? Debido a Any tipo de retorno en la declaración de protocolo.

Al usar los genéricos, puede evitar este problema que puede causar errores de tiempo de

ejecución (¡y no queremos tenerlos!)

```
protocol JSONDecodable {
    associatedtype Element
    static func from(_ json: [String: Any]) -> Element?
}

struct TestObject: JSONDecodable {
    static func from(_ json: [String: Any]) -> TestObject? {
    }
}

let testObject = TestObject.from(myJson) // testObject is now automatically `TestObject?`
```

Restricciones de tipo avanzado

Es posible especificar varias restricciones de tipo para genéricos usando la cláusula where :

```
func doSomething<T where T: Comparable, T: Hashable>(first: T, second: T) {
    // Access hashable function
    guard first.hashValue == second.hashValue else {
        return
    }
    // Access comparable function
    if first == second {
        print("\(first) and \(second) are equal.")
    }
}
```

También es válido escribir la cláusula where después de la lista de argumentos:

```
func doSomething<T>(first: T, second: T) where T: Comparable, T: Hashable {
    // Access hashable function
    guard first.hashValue == second.hashValue else {
        return
    }
    // Access comparable function
    if first == second {
        print("\(first) and \(second) are equal.")
    }
}
```

Las extensiones se pueden restringir a tipos que satisfacen condiciones. La función solo está disponible para instancias que cumplan con las condiciones de tipo:

```
// "Element" is the generics type defined by "Array". For this example, we
// want to add a function that requires that "Element" can be compared, that
// is: it needs to adhere to the Equatable protocol.
public extension Array where Element: Equatable {
    /// Removes the given object from the array.
    mutating func remove(_ element: Element) {
        // We could also use "self.index(of: element)" here, as "index(of:_)"
        // is also defined in an extension with "where Element: Equatable".
        // For the sake of this example, explicitly make use of the Equatable.
        if let index = self.index(where: { $0 == element }) {
            self.remove(at: index)
        } else {
            fatalError("Removal error, no such element:\"\(element)\" in array.\n")
        }
```

```
    }
}
```

Lea Genéricos en línea: https://riptutorial.com/es/swift/topic/774/genericos

Capítulo 31: Gestión de la memoria

Introducción

Este tema describe cómo y cuándo el tiempo de ejecución de Swift asignará memoria para las estructuras de datos de la aplicación y cuándo se recuperará esa memoria. De forma predeterminada, las instancias de la clase de respaldo de memoria se administran a través del conteo de referencias. Las estructuras se pasan siempre a través de la copia. Para excluirse del esquema de administración de memoria incorporado, se podría usar la estructura [Unmanaged] [1]. [1]: https://developer.apple.com/reference/swift/unmanaged

Observaciones

Cuándo usar la palabra clave débil:

La palabra clave weak debe usarse, si un objeto referenciado puede ser desasignado durante la vida útil del objeto que contiene la referencia.

Cuándo usar la palabra clave sin dueño:

Se debe usar la palabra clave unowned propietario, si no se espera que un objeto referenciado se desasigne durante la vida útil del objeto que contiene la referencia.

Escollos

Un error frecuente es olvidarse de crear referencias a objetos, que se requieren para vivir después de que finaliza una función, como administradores de ubicación, administradores de movimiento, etc.

Ejemplo:

```
class A : CLLocationManagerDelegate
{
    init()
    {
        let locationManager = CLLocationManager()
        locationManager.delegate = self
        locationManager.startLocationUpdates()
    }
}
```

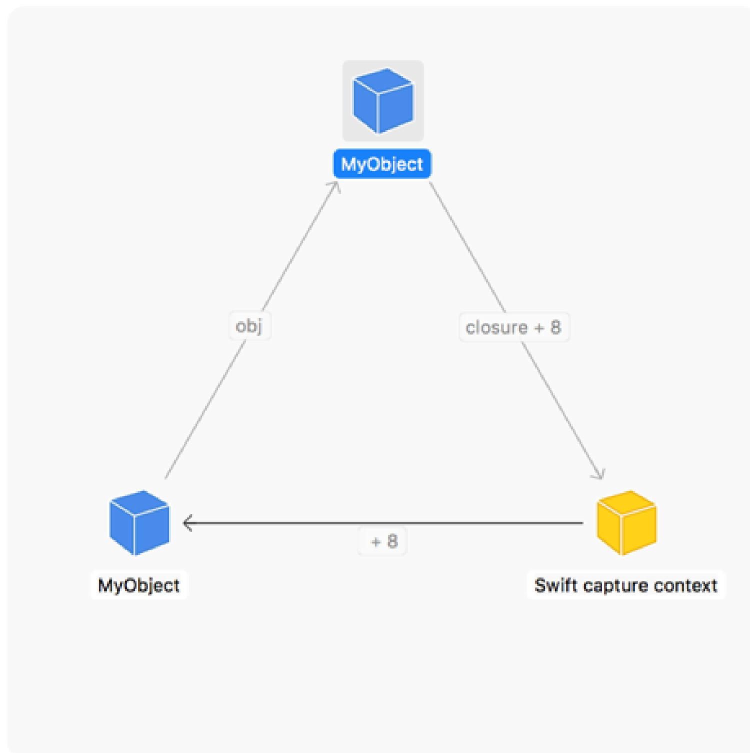
Este ejemplo no funcionará correctamente, ya que el administrador de ubicación se desasigna una vez que el inicializador regresa. La solución adecuada es crear una referencia fuerte como una variable de instancia:

```
class A : CLLocationManagerDelegate
{
    let locationManager:CLLocationManager

    init()
    {
        locationManager = CLLocationManager()
        locationManager.delegate = self
        locationManager.startLocationUpdates()
    }
}
```

Examples

Ciclos de referencia y referencias débiles

Un *ciclo de referencia* (o *ciclo de retención*) se llama así porque indica un ciclo en el gráfico de objetos :



Cada flecha indica un objeto que retiene otro (una referencia fuerte). A menos que el ciclo se rompa, la memoria de estos objetos **nunca se liberará** .

Se crea un ciclo de retención cuando dos instancias de clases se hacen referencia:

```
class A { var b: B? = nil }
class B { var a: A? = nil }

let a = A()
let b = B()

a.b = b  // a retains b
b.a = a  // b retains a -- a reference cycle
```

Ambas instancias vivirán hasta que el programa termine. Este es un ciclo de retención.

Referencias débiles

Para evitar los ciclos de retención, use la palabra clave weak o unowned al crear referencias para romper los ciclos de retención.

```
class B { weak var a: A? = nil }
```

Las referencias débiles o sin propiedad no aumentarán el recuento de referencias de una instancia. Estas referencias no contribuyen a retener los ciclos. La referencia débil se **vuelve nil** cuando el objeto al que hace referencia se desasigna.

```
a.b = b  // a retains b
b.a = a  // b holds a weak reference to a -- not a reference cycle
```

Cuando trabaje con cierres, también puede usar weak y unowned en las listas de captura .

Gestión de memoria manual

Al interactuar con las API de C, es posible que desee retroceder el contador de referencia de Swift. Al hacerlo se logra con objetos no gestionados.

Si necesita suministrar un puntero de tipo punted a una función C, use el método de toOpaque de la estructura no Unmanaged para obtener un puntero sin fromOpaque y de fromOpaque para recuperar la instancia original:

```
setupDisplayLink() {
  let pointerToSelf: UnsafeRawPointer = Unmanaged.passUnretained(self).toOpaque()
  CVDisplayLinkSetOutputCallback(self.displayLink, self.redraw, pointerToSelf)
}

func redraw(pointerToSelf: UnsafeRawPointer, /* args omitted */) {
  let recoveredSelf = Unmanaged<Self>.fromOpaque(pointerToSelf).takeUnretainedValue()
  recoveredSelf.doRedraw()
}
```

Tenga en cuenta que, si utiliza passUnretained y sus homólogos, es necesario tomar todas las precauciones como con las referencias unowned .

Para interactuar con las API heredadas de Objective-C, uno podría querer afectar manualmente el conteo de referencias de un determinado objeto. Para eso Unmanaged tiene métodos respectivos de retain y release . No obstante, es más deseable usar passRetained y takeRetainedValue , que realizan la retención antes de devolver el resultado:

```
func preferredFilenameExtension(for uti: String) -> String! {
  let result = UTTypeCopyPreferredTagWithClass(uti, kUTTagClassFilenameExtension)
  guard result != nil else { return nil }

  return result!.takeRetainedValue() as String
}
```

Estas soluciones siempre deben ser el último recurso, y las API nativas de idioma siempre deben ser preferidas.

Lea Gestión de la memoria en línea: https://riptutorial.com/es/swift/topic/745/gestion-de-la-memoria

Capítulo 32: Gestor de paquetes Swift

Examples

Creación y uso de un simple paquete Swift

Para crear un paquete Swift, abra un terminal y luego cree una carpeta vacía:

```
mkdir AwesomeProject
cd AwesomeProject
```

E inicie un repositorio Git:

```
git init
```

Luego crea el paquete en sí. Uno podría crear la estructura del paquete manualmente, pero hay una forma sencilla de usar el comando CLI.

Si quieres hacer un ejecutable:

```
swift package init --type executable
```

Se generarán varios archivos. Entre ellos, *main.swift* será el punto de entrada para su aplicación.

Si quieres hacer una biblioteca:

```
swift package init --type library
```

El archivo *AwesomeProject.swift* generado se utilizará como el archivo principal de esta biblioteca.

En ambos casos, puede agregar otros archivos Swift en la carpeta *Orígenes* (se aplican las reglas habituales para el control de acceso).

El archivo *Package.swift se llenará* automáticamente con este contenido:

```
import PackageDescription

let package = Package(
    name: "AwesomeProject"
)
```

La versión del paquete se realiza con etiquetas Git:

```
git tag '1.0.0'
```

Una vez que se envía a un repositorio Git local o remoto, su paquete estará disponible para otros proyectos.

Su paquete ya está listo para ser compilado:

```
swift build
```

El proyecto compilado estará disponible en la carpeta *.build / debug* .

Su propio paquete también puede resolver dependencias a otros paquetes. Por ejemplo, si desea incluir "SomeOtherPackage" en su propio proyecto, cambie su archivo *Package.swift* para incluir

la dependencia:

```
import PackageDescription

let package = Package(
    name: "AwesomeProject",
    targets: [],
    dependencies: [
        .Package(url: "https://github.com/someUser/SomeOtherPackage.git",
                 majorVersion: 1),
    ]
)
```

Luego, vuelva a construir su proyecto: Swift Package Manager resolverá, descargará y construirá las dependencias automáticamente.

Lea Gestor de paquetes Swift en línea: https://riptutorial.com/es/swift/topic/5144/gestor-de-paquetes-swift

Capítulo 33: Hash criptográfico

Examples

MD2, MD4, MD5, SHA1, SHA224, SHA256, SHA384, SHA512 (Swift 3)

> Estas funciones incluyen el hash String o la entrada de datos con uno de los ocho algoritmos de hash criptográficos.

El parámetro de nombre especifica el nombre de la función hash como una cadena
Las funciones admitidas son MD2, MD4, MD5, SHA1, SHA224, SHA256, SHA384 y SHA512

Este ejemplo requiere Crypto común
Es necesario tener un encabezado puente al proyecto:
#import <CommonCrypto/CommonCrypto.h>
Agregue el Security.framework al proyecto.

Esta función toma un nombre de hash y los datos que se van a hash y devuelve un dato:

```
name: A name of a hash function as a String
data: The Data to be hashed
returns: the hashed result as Data
```

```
func hash(name:String, data:Data) -> Data? {
    let algos = ["MD2":    (CC_MD2,    CC_MD2_DIGEST_LENGTH),
                 "MD4":    (CC_MD4,    CC_MD4_DIGEST_LENGTH),
                 "MD5":    (CC_MD5,    CC_MD5_DIGEST_LENGTH),
                 "SHA1":   (CC_SHA1,   CC_SHA1_DIGEST_LENGTH),
                 "SHA224": (CC_SHA224, CC_SHA224_DIGEST_LENGTH),
                 "SHA256": (CC_SHA256, CC_SHA256_DIGEST_LENGTH),
                 "SHA384": (CC_SHA384, CC_SHA384_DIGEST_LENGTH),
                 "SHA512": (CC_SHA512, CC_SHA512_DIGEST_LENGTH)]
    guard let (hashAlgorithm, length) = algos[name]  else { return nil }
    var hashData = Data(count: Int(length))

    _ = hashData.withUnsafeMutableBytes {digestBytes in
        data.withUnsafeBytes {messageBytes in
            hashAlgorithm(messageBytes, CC_LONG(data.count), digestBytes)
        }
    }
    return hashData
}
```

Esta función toma un nombre de hash y una cadena para ser hash y devuelve un dato:

```
name: A name of a hash function as a String
string: The String to be hashed
returns: the hashed result as Data
```

```
func hash(name:String, string:String) -> Data? {
    let data = string.data(using:.utf8)!
    return hash(name:name, data:data)
}
```

Ejemplos:

```
let clearString = "clearData0123456"
let clearData   = clearString.data(using:.utf8)!
print("clearString: \(clearString)")
print("clearData: \(clearData as NSData)")

let hashSHA256 = hash(name:"SHA256", string:clearString)
print("hashSHA256: \(hashSHA256! as NSData)")

let hashMD5 = hash(name:"MD5", data:clearData)
print("hashMD5: \(hashMD5! as NSData)")
```

Salida:

```
clearString: clearData0123456
clearData: <636c6561 72446174 61303132 33343536>

hashSHA256: <aabc766b 6b357564 e41f4f91 2d494bcc bfa16924 b574abbd ba9e3e9d a0c8920a>
hashMD5: <4df665f7 b94aea69 695b0e7b baf9e9d6>
```

HMAC con MD5, SHA1, SHA224, SHA256, SHA384, SHA512 (Swift 3)

> Estas funciones incluyen el hash String o la entrada de datos con uno de los ocho algoritmos de hash criptográficos.

El parámetro name especifica el nombre de la función hash como una cadena. Las funciones admitidas son MD5, SHA1, SHA224, SHA256, SHA384 y SHA512

Este ejemplo requiere Crypto común
Es necesario tener un encabezado puente al proyecto:
#import <CommonCrypto/CommonCrypto.h>
Agregue el Security.framework al proyecto.

Estas funciones toman un nombre de hash, un mensaje que se va a hash, una clave y devuelven un resumen:

```
hashName: name of a hash function as String
message:  message as Data
key:      key as Data
returns:  digest as Data
```

```
func hmac(hashName:String, message:Data, key:Data) -> Data? {
    let algos = ["SHA1":   (kCCHmacAlgSHA1,   CC_SHA1_DIGEST_LENGTH),
                 "MD5":    (kCCHmacAlgMD5,    CC_MD5_DIGEST_LENGTH),
                 "SHA224": (kCCHmacAlgSHA224, CC_SHA224_DIGEST_LENGTH),
                 "SHA256": (kCCHmacAlgSHA256, CC_SHA256_DIGEST_LENGTH),
                 "SHA384": (kCCHmacAlgSHA384, CC_SHA384_DIGEST_LENGTH),
                 "SHA512": (kCCHmacAlgSHA512, CC_SHA512_DIGEST_LENGTH)]
    guard let (hashAlgorithm, length) = algos[hashName]  else { return nil }
    var macData = Data(count: Int(length))

    macData.withUnsafeMutableBytes {macBytes in
        message.withUnsafeBytes {messageBytes in
            key.withUnsafeBytes {keyBytes in
                CCHmac(CCHmacAlgorithm(hashAlgorithm),
                       keyBytes,     key.count,
                       messageBytes, message.count,
                       macBytes)
            }
```

```
        }
    }
    return macData
}
```

```
hashName: name of a hash function as String
message:  message as String
key:      key as String
returns:  digest as Data
```

```
func hmac(hashName:String, message:String, key:String) -> Data? {
    let messageData = message.data(using:.utf8)!
    let keyData = key.data(using:.utf8)!
    return hmac(hashName:hashName, message:messageData, key:keyData)
}
```

```
hashName: name of a hash function as String
message:  message as String
key:      key as Data
returns:  digest as Data
```

```
func hmac(hashName:String, message:String, key:Data) -> Data? {
    let messageData = message.data(using:.utf8)!
    return hmac(hashName:hashName, message:messageData, key:key)
}
```

// Ejemplos

```
let clearString = "clearData0123456"
let keyString   = "keyData8901234562"
let clearData   = clearString.data(using:.utf8)!
let keyData     = keyString.data(using:.utf8)!
print("clearString: \(clearString)")
print("keyString:   \(keyString)")
print("clearData: \(clearData as NSData)")
print("keyData:   \(keyData as NSData)")

let hmacData1 = hmac(hashName:"SHA1", message:clearData, key:keyData)
print("hmacData1: \(hmacData1! as NSData)")

let hmacData2 = hmac(hashName:"SHA1", message:clearString, key:keyString)
print("hmacData2: \(hmacData2! as NSData)")

let hmacData3 = hmac(hashName:"SHA1", message:clearString, key:keyData)
print("hmacData3: \(hmacData3! as NSData)")
```

Salida:

```
clearString: clearData0123456
keyString:   keyData8901234562
clearData: <636c6561 72446174 61303132 33343536>
keyData:   <6b657944 61746138 39303132 33343536 32>

hmacData1: <bb358f41 79b68c08 8e93191a da7dabbc 138f2ae6>
hmacData2: <bb358f41 79b68c08 8e93191a da7dabbc 138f2ae6>
```

```
hmacData3: <bb358f41 79b68c08 8e93191a da7dabbc 138f2ae6>
```

Lea Hash criptográfico en línea: https://riptutorial.com/es/swift/topic/7885/hash-criptografico

Capítulo 34: Inicializadores

Examples

Establecer valores de propiedad predeterminados

Puede usar un inicializador para establecer valores de propiedad predeterminados:

```
struct Example {
    var upvotes: Int
    init() {
        upvotes = 42
    }
}
let myExample = Example() // call the initializer
print(myExample.upvotes) // prints: 42
```

O, especifique valores de propiedad predeterminados como parte de la declaración de la propiedad:

```
struct Example {
    var upvotes = 42 // the type 'Int' is inferred here
}
```

Las clases y las estructuras **deben** establecer todas las propiedades almacenadas en un valor inicial adecuado en el momento en que se crea una instancia. Este ejemplo no se compilará, porque el inicializador no dio un valor inicial para downvotes :

```
struct Example {
    var upvotes: Int
    var downvotes: Int
    init() {
         upvotes = 0
    } // error: Return from initializer without initializing all stored properties
}
```

Personalización de inicialización con paramatizadores.

```
struct MetricDistance {
    var distanceInMeters: Double

    init(fromCentimeters centimeters: Double) {
        distanceInMeters = centimeters / 100
    }
    init(fromKilometers kilos: Double) {
        distanceInMeters = kilos * 1000
    }
}

let myDistance = MetricDistance(fromCentimeters: 42)
// myDistance.distanceInMeters is 0.42
let myOtherDistance = MetricDistance(fromKilometers: 42)
// myOtherDistance.distanceInMeters is 42000
```

Tenga en cuenta que no puede omitir las etiquetas de parámetros:

```
let myBadDistance = MetricDistance(42) // error: argument labels do not match any available
```

```
overloads
```

Para permitir la omisión de etiquetas de parámetros, use un guión bajo _ como la etiqueta:

```
struct MetricDistance {
    var distanceInMeters: Double
    init(_ meters: Double) {
        distanceInMeters = meters
    }
}
let myDistance = MetricDistance(42) // distanceInMeters = 42
```

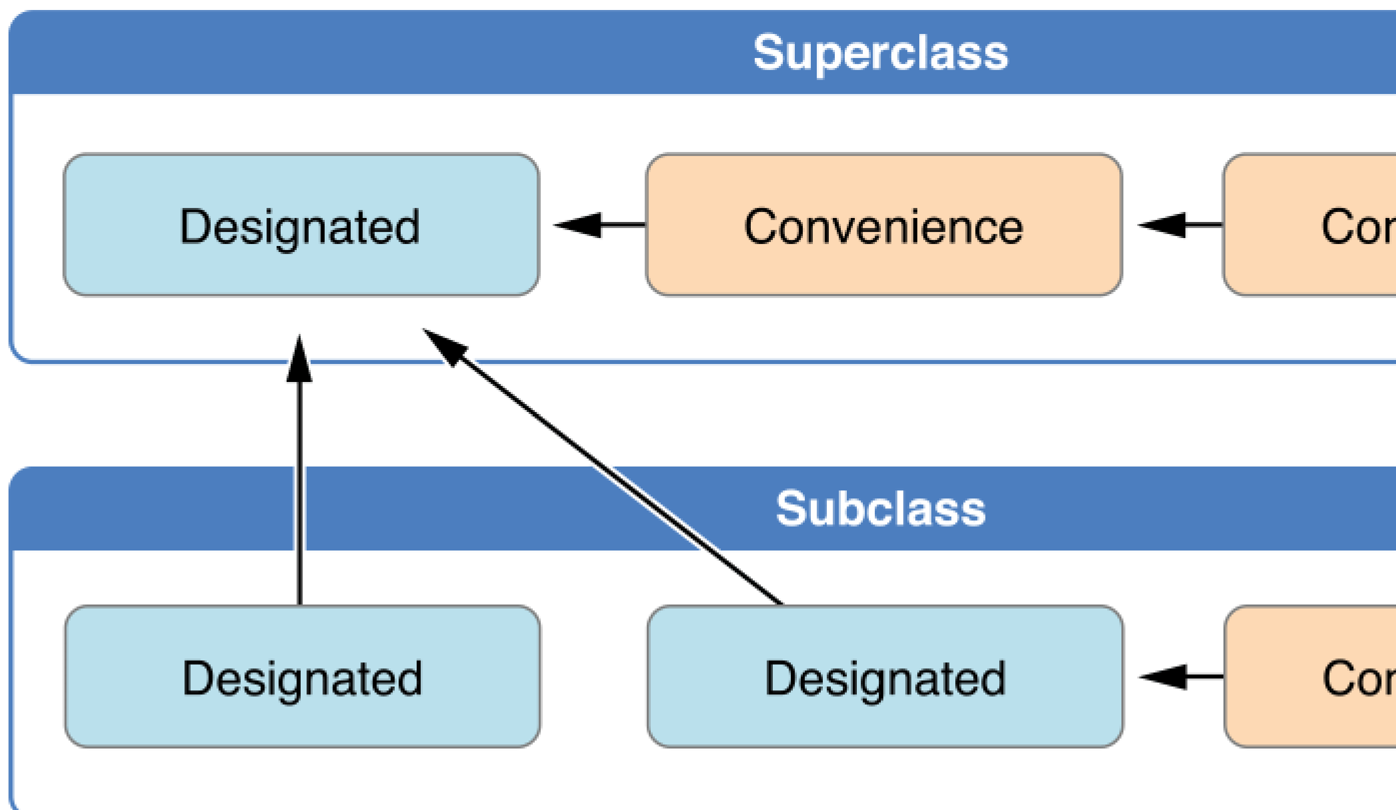
Si las etiquetas de sus argumentos comparten nombres con una o más propiedades, use self para establecer explícitamente los valores de propiedad:

```
struct Color {
    var red, green, blue: Double
    init(red: Double, green: Double, blue: Double) {
        self.red = red
        self.green = green
        self.blue = blue
    }
}
```

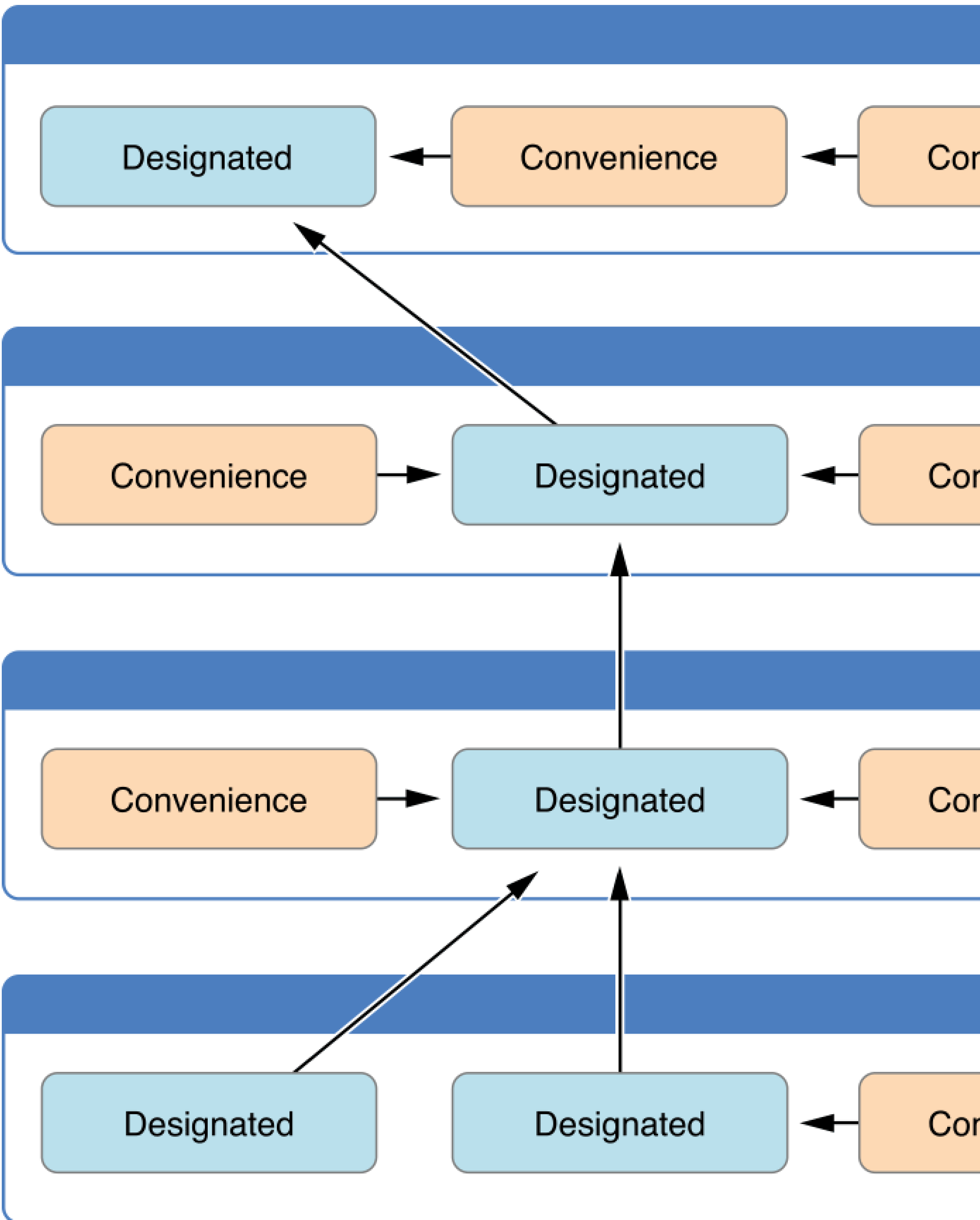
Conveniencia inic

Las clases de Swift admiten tener múltiples formas de inicializarse. Siguiendo las especificaciones de Apple, estas 3 reglas deben ser respetadas:

1. Un inicializador designado debe llamar a un inicializador designado desde su superclase inmediata.



2. Un inicializador de conveniencia debe llamar a otro inicializador de la misma clase.
3. Un inicializador de conveniencia debe finalmente llamar a un inicializador designado.




```
class Foo {

    var someString: String
    var someValue: Int
    var someBool: Bool
```

```
    // Designated Initializer
    init(someString: String, someValue: Int, someBool: Bool)
    {
        self.someString = someString
        self.someValue = someValue
        self.someBool = someBool
    }

    // A convenience initializer must call another initializer from the same class.
    convenience init()
    {
        self.init(otherString: "")
    }

    // A convenience initializer must ultimately call a designated initializer.
convenience init(otherString: String)
    {
        self.init(someString: otherString, someValue:  0, someBool: false)
    }
}


class Baz: Foo
{
    var someFloat: Float

    // Designed initializer
    init(someFloat: Float)
    {
        self.someFloat = someFloat

        // A designated initializer must call a designated initializer from its immediate
superclass.
        super.init(someString: "", someValue: 0, someBool: false)
    }

    // A convenience initializer must call another initializer from the same class.
    convenience init()
    {
        self.init(someFloat: 0)
    }
}
```

Inicializador designado

```
let c = Foo(someString: "Some string", someValue: 10, someBool: true)
```

Inicia conveniencia ()

```
let a = Foo()
```

Convenience init (otherString: String)

```
let b = Foo(otherString: "Some string")
```

Inicializador designado (llamará al inicializador designado de superclase)

```
let d = Baz(someFloat: 3)
```

Inicia conveniencia ()

```
let e = Baz()
```

Fuente de la imagen: The Swift Programming Languag e

Iniciador Throwable

Uso del manejo de errores para hacer el inicializador Struct (o clase) como inicializador lanzable:

Ejemplo de manejo de errores enumeración:

```
enum ValidationError: Error {
    case invalid
}
```

Puede usar la enumeración de manejo de errores para verificar que el parámetro de Struct (o clase) cumpla con el requisito esperado

```
struct User {
    let name: String

    init(name: String?) throws {

        guard let name = name else {
           ValidationError.invalid
        }

        self.name = name
    }
}
```

Ahora, puedes usar inicializador lanzable por:

```
do {
   let user = try User(name: "Sample name")

   // success
}
catch ValidationError.invalid {
     // handle error
}
```

Lea Inicializadores en línea: https://riptutorial.com/es/swift/topic/1778/inicializadores

Capítulo 35: Inyección de dependencia

Examples

Inyección de dependencia con controladores de vista

Introducción a la inyección dependiente

Una aplicación se compone de muchos objetos que colaboran entre sí. Los objetos suelen depender de otros objetos para realizar alguna tarea. Cuando un objeto es responsable de hacer referencia a sus propias dependencias, conduce a un código altamente acoplado, difícil de probar y difícil de cambiar.

La inyección de dependencia es un patrón de diseño de software que implementa la inversión de control para resolver dependencias. Una inyección es el paso de la dependencia a un objeto dependiente que la usaría. Esto permite una separación de las dependencias del cliente del comportamiento del cliente, lo que permite que la aplicación se acople de forma flexible.

No debe confundirse con la definición anterior: una inyección de dependencia simplemente significa darle a un objeto sus variables de instancia.

Es así de simple, pero proporciona muchos beneficios:

- Más fácil probar su código (utilizando pruebas automatizadas como pruebas de unidad y UI)
- cuando se usa en tándem con programación orientada a protocolos, facilita la modificación de la implementación de una clase determinada, más fácil de refactorizar
- Hace que el código sea más modular y reutilizable.

Hay tres formas más comunes de implementar la inyección de dependencia (DI) en una aplicación:

1. Inyección de inicializador
2. Inyección de propiedad
3. Uso de marcos DI de terceros (como Swinject, Cleanse, Dip o Typhoon)

Hay un artículo interesante con enlaces a más artículos acerca de la inyección de dependencia, así que verifique si desea profundizar en el principio de DI e Inversión de control.

Vamos a mostrar cómo usar DI con controladores de vista: una tarea diaria para un desarrollador de iOS promedio.

Ejemplo sin DI

Tendremos dos controladores de vista: **LoginViewController** y **TimelineViewController** . LoginViewController se utiliza para iniciar sesión y, una vez que se ha iniciado correctamente, cambiará al TimelineViewController. Ambos controladores de vista dependen de **FirebaseNetworkService** .

LoginViewController

```
class LoginViewController: UIViewController {

    var networkService = FirebaseNetworkService()

    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

TimelineViewController

```
class TimelineViewController: UIViewController {

    var networkService = FirebaseNetworkService()

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    @IBAction func logoutButtonPressed(_ sender: UIButton) {
        networkService.logutCurrentUser()
    }
}
```

FirebaseNetworkService

```
class FirebaseNetworkService {

    func loginUser(username: String, passwordHash: String) {
        // Implementation not important for this example
    }

    func logutCurrentUser() {
        // Implementation not important for this example
    }
}
```

Este ejemplo es muy simple, pero supongamos que tiene 10 o 15 controladores de vista diferentes y algunos de ellos también dependen de FirebaseNetworkService. En algún momento, usted desea cambiar Firebase como su servicio backend con el servicio interno de su compañía. Para hacer eso, tendrá que pasar por cada controlador de vista y cambiar FirebaseNetworkService con CompanyNetworkService. Y si algunos de los métodos en CompanyNetworkService han cambiado, tendrá mucho trabajo por hacer.

Las pruebas de unidad y UI no son el alcance de este ejemplo, pero si quisiera probar en unidad los controladores de vista con dependencias estrechamente acopladas, sería muy difícil hacerlo.

Reescribamos este ejemplo e inyectemos el servicio de red a nuestros controladores de visualización.

Ejemplo con inyección de dependencia

Para aprovechar al máximo la inyección de dependencia, definamos la funcionalidad del servicio de red en un protocolo. De esta manera, los controladores de vista que dependen de un servicio de red ni siquiera tendrán que conocer la implementación real de este.

```
protocol NetworkService {
    func loginUser(username: String, passwordHash: String)
    func logutCurrentUser()
}
```

Agregue una implementación del protocolo NetworkService:

```
class FirebaseNetworkServiceImpl: NetworkService {
    func loginUser(username: String, passwordHash: String) {
        // Firebase implementation
    }

    func logutCurrentUser() {
        // Firebase implementation
    }
```

```
}
```

Cambiemos LoginViewController y TimelineViewController para utilizar el nuevo protocolo NetworkService en lugar de FirebaseNetworkService.

LoginViewController

```
class LoginViewController: UIViewController {

    // No need to initialize it here since an implementation
    // of the NetworkService protocol will be injected
    var networkService: NetworkService?

    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

TimelineViewController

```
class TimelineViewController: UIViewController {

    var networkService: NetworkService?

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    @IBAction func logoutButtonPressed(_ sender: UIButton) {
        networkService?.logutCurrentUser()
    }
}
```

Ahora, la pregunta es: ¿Cómo inyectamos la implementación correcta de NetworkService en LoginViewController y TimelineViewController?

Dado que LoginViewController es el controlador de vista de inicio y se mostrará cada vez que se inicie la aplicación, podemos inyectar todas las dependencias en **AppDelegate** .

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
    // This logic will be different based on your project's structure or whether
    // you have a navigation controller or tab bar controller for your starting view
controller
    if let loginVC = window?.rootViewController as? LoginViewController {
        loginVC.networkService = FirebaseNetworkServiceImpl()
    }
    return true
}
```

En el AppDelegate simplemente estamos tomando la referencia al primer controlador de vista (LoginViewController) e inyectando la implementación de NetworkService usando el método de inyección de propiedad.

Ahora, la siguiente tarea es inyectar la implementación de NetworkService en el TimelineViewController. La forma más fácil es hacerlo cuando LoginViewController está en transición a TimlineViewController.

Agregaremos el código de inyección en el método prepareForSegue en el LoginViewController (si está utilizando un método diferente para navegar a través de los controladores de vista, coloque el código de inyección allí).

Nuestra clase LoginViewController se ve así ahora:

```
class LoginViewController: UIViewController {
    // No need to initialize it here since an implementation
    // of the NetworkService protocol will be injected
    var networkService: NetworkService?

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if segue.identifier == "TimelineViewController" {
            if let timelineVC = segue.destination as? TimelineViewController {
                // Injecting the NetworkService implementation
                timelineVC.networkService = networkService
            }
        }
    }
}
```

Hemos terminado y es así de fácil.

Ahora imagine que queremos cambiar nuestra implementación de NetworkService de Firebase a la implementación backend personalizada de nuestra empresa. Todo lo que tendríamos que hacer es:

Agregar nueva clase de implementación de NetworkService:

```
class CompanyNetworkServiceImpl: NetworkService {
    func loginUser(username: String, passwordHash: String) {
        // Company API implementation
    }

    func logutCurrentUser() {
        // Company API implementation
    }
}
```

Cambie FirebaseNetworkServiceImpl con la nueva implementación en AppDelegate:

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        // This logic will be different based on your project's structure or whether
        // you have a navigation controller or tab bar controller for your starting view
controller
        if let loginVC = window?.rootViewController as? LoginViewController {
            loginVC.networkService = CompanyNetworkServiceImpl()
        }
        return true
    }
```

Eso es todo, hemos cambiado toda la implementación subyacente del protocolo NetworkService sin siquiera tocar LoginViewController o TimelineViewController.

Como este es un ejemplo simple, es posible que no vea todos los beneficios en este momento, pero si intenta usar DI en sus proyectos, verá los beneficios y siempre usará la inyección de dependencia.

Tipos de inyección de dependencia

Este ejemplo mostrará cómo usar el patrón de diseño de inyección de dependencia (**DI**) en Swift

usando estos métodos:

1. **Inyección de inicializador** (el término correcto es Inyección de constructor, pero como Swift tiene inicializadores, se llama inyección de inicializador)
2. **Inyección de propiedad**
3. **Método de inyección**

Ejemplo de configuración sin DI

```
    protocol Engine {
        func startEngine()
        func stopEngine()
    }

    class TrainEngine: Engine {
        func startEngine() {
            print("Engine started")
        }

        func stopEngine() {
            print("Engine stopped")
        }
    }

    protocol TrainCar {
    var numberOfSeats: Int { get }
    func attachCar(attach: Bool)
}

class RestaurantCar: TrainCar {
    var numberOfSeats: Int {
        get {
            return 30
        }
    }
    func attachCar(attach: Bool) {
        print("Attach car")
    }
}

class PassengerCar: TrainCar {
    var numberOfSeats: Int {
        get {
            return 50
        }
    }
    func attachCar(attach: Bool) {
        print("Attach car")
    }
}

class Train {
    let engine: Engine?
    var mainCar: TrainCar?
}
```

Inyección de dependencia del inicializador

Como su nombre lo indica, todas las dependencias se inyectan a través del inicializador de clase. Para inyectar dependencias a través del inicializador, agregaremos el inicializador a la

clase Train .

La clase de tren ahora se ve así:

```
class Train {
    let engine: Engine?
    var mainCar: TrainCar?

    init(engine: Engine) {
        self.engine = engine
    }
}
```

Cuando deseamos crear una instancia de la clase Train, usaremos el inicializador para inyectar una implementación de Motor específica:

```
let train = Train(engine: TrainEngine())
```

NOTA: La principal ventaja de la inyección de inicializador frente a la inyección de propiedades es que podemos establecer la variable como variable privada o incluso convertirla en una constante con la palabra clave let (como hicimos en nuestro ejemplo). De esta manera podemos asegurarnos de que nadie pueda acceder o cambiarlo.

Propiedades Dependencia Inyección

DI usando propiedades es incluso más simple que usar un inicializador. Inyectemos una dependencia de PassengerCar en el objeto de tren que ya creamos utilizando las propiedades DI:

```
train.mainCar = PassengerCar()
```

Eso es. mainCar nuestro tren ahora es una instancia de PassengerCar .

Método de inyección de dependencia

Este tipo de inyección de dependencia es un poco diferente a los dos anteriores porque no afectará a todo el objeto, pero solo inyectará una dependencia que se utilizará en el alcance de un método específico. Cuando una dependencia solo se usa en un solo método, generalmente no es bueno hacer que todo el objeto dependa de él. Agreguemos un nuevo método a la clase de tren:

```
func reparkCar(trainCar: TrainCar) {
    trainCar.attachCar(attach: true)
    engine?.startEngine()
    engine?.stopEngine()
    trainCar.attachCar(attach: false)
}
```

Ahora, si llamamos al nuevo método de clase de Train, inyectaremos el TrainCar utilizando el método de inyección de dependencia.

```
train.reparkCar(trainCar: RestaurantCar())
```

Lea Inyección de dependencia en línea: https://riptutorial.com/es/swift/topic/8198/inyeccion-de-dependencia

Capítulo 36: La declaración diferida

Examples

Cuándo usar una declaración diferida

Una declaración defer consiste en un bloque de código, que se ejecutará cuando una función regrese y se debe usar para la limpieza.

Como las declaraciones de los guard de Swift fomentan un estilo de retorno temprano, pueden existir muchos caminos posibles para un retorno. Una declaración defer proporciona un código de limpieza, que luego no necesita repetirse cada vez.

También puede ahorrar tiempo durante la depuración y la creación de perfiles, ya que se pueden evitar las pérdidas de memoria y los recursos abiertos no utilizados debido a una limpieza olvidada.

Se puede usar para desasignar un búfer al final de una función:

```
func doSomething() {
    let data = UnsafeMutablePointer<UInt8>(allocatingCapacity: 42)
    // this pointer would not be released when the function returns
    // so we add a defer-statement
    defer {
        data.deallocateCapacity(42)
    }
    // it will be executed when the function returns.

    guard condition else {
        return /* will execute defer-block */
    }

}   // The defer-block will also be executed on the end of the function.
```

También se puede usar para cerrar recursos al final de una función:

```
func write(data: UnsafePointer<UInt8>, dataLength: Int) throws {
    var stream:NSOutputStream = getOutputStream()
    defer {
        stream.close()
    }

    let written = stream.write(data, maxLength: dataLength)
    guard written >= 0 else {
        throw stream.streamError! /* will execute defer-block */
    }

}    // the defer-block will also be executed on the end of the function
```

Cuando NO usar una declaración diferida

Cuando use una declaración diferida, asegúrese de que el código se mantenga legible y que el orden de ejecución quede claro. Por ejemplo, el siguiente uso de la declaración diferida hace que el orden de ejecución y la función del código sean difíciles de comprender.

```
postfix func ++ (inout value: Int) -> Int {
    defer { value += 1 } // do NOT do this!
    return value
}
```

Lea La declaración diferida en línea: https://riptutorial.com/es/swift/topic/4932/la-declaracion-diferida

Capítulo 37: Las clases

Observaciones

El init() es un método especial en clases que se utiliza para declarar un inicializador para la clase. Más información se puede encontrar aquí: Inicializadores

Examples

Definiendo una clase

Se define una clase como esta:

```
class Dog {}
```

Una clase también puede ser una subclase de otra clase:

```
class Animal {}
class Dog: Animal {}
```

En este ejemplo, Animal también podría ser un protocolo que cumple Dog .

Semántica de referencia

Las clases son **tipos de referencia** , lo que significa que varias variables pueden referirse a la misma instancia.

```
class Dog {
    var name = ""
}

let firstDog = Dog()
firstDog.name = "Fido"

let otherDog = firstDog  // otherDog points to the same Dog instance
otherDog.name = "Rover"  // modifying otherDog also modifies firstDog

print(firstDog.name)  // prints "Rover"
```

Debido a que las clases son tipos de referencia, incluso si la clase es una constante, sus propiedades variables pueden modificarse.

```
class Dog {
    var name: String // name is a variable property.
    let age: Int // age is a constant property.
    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}

let constantDog = Dog(name: "Rover", age: 5)// This instance is a constant.
var variableDog = Dog(name: "Spot", age 7)// This instance is a variable.

constantDog.name = "Fido" // Not an error because name is a variable property.
constantDog.age = 6 // Error because age is a constant property.
constantDog = Dog(name: "Fido", age: 6)
```

```
/* The last one is an error because you are changing the actual reference, not
just what the reference points to. */

variableDog.name = "Ace" // Not an error because name is a variable property.
variableDog.age = 8 // Error because age is a constant property.
variableDog = Dog(name: "Ace", age: 8)
/* The last one is not an error because variableDog is a variable instance and
therefore the actual reference can be changed. */
```

Probar si dos objetos son *idénticos* (apuntan a la misma instancia exacta) usando `===` :

```
class Dog: Equatable {
    let name: String
    init(name: String) { self.name = name }
}

// Consider two dogs equal if their names are equal.
func ==(lhs: Dog, rhs: Dog) -> Bool {
    return lhs.name == rhs.name
}

// Create two Dog instances which have the same name.
let spot1 = Dog(name: "Spot")
let spot2 = Dog(name: "Spot")

spot1 == spot2   // true, because the dogs are equal
spot1 != spot2   // false

spot1 === spot2  // false, because the dogs are different instances
spot1 !== spot2  // true
```

Propiedades y metodos

Las clases pueden definir propiedades que pueden usar las instancias de la clase. En este ejemplo, `Dog` tiene dos propiedades: `name` y `dogYearAge` :

```
class Dog {
    var name = ""
    var dogYearAge = 0
}
```

Puede acceder a las propiedades con sintaxis de puntos:

```
let dog = Dog()
print(dog.name)
print(dog.dogYearAge)
```

Las clases también pueden definir métodos que se pueden llamar en las instancias, se declaran similares a las funciones normales, solo dentro de la clase:

```
class Dog {
    func bark() {
        print("Ruff!")
    }
}
```

Los métodos de llamada también usan la sintaxis de puntos:

```
dog.bark()
```

Clases y herencia múltiple

Swift no admite la herencia múltiple. Es decir, no puede heredar de más de una clase.

```
class Animal { ... }
class Pet { ... }

class Dog: Animal, Pet { ... } // This will result in a compiler error.
```

En su lugar, se recomienda utilizar la composición al crear sus tipos. Esto se puede lograr mediante el uso de protocolos .

deinit

```
class ClassA {

    var timer: NSTimer!

    init() {
        // initialize timer
    }

    deinit {
        // code
        timer.invalidate()
    }
}
```

Lea Las clases en línea: https://riptutorial.com/es/swift/topic/459/las-clases

Capítulo 38: Lectura y escritura JSON

Sintaxis

- NSJSONSerialization.JSONObjectWithData (jsonData, opciones: NSJSONReadingOptions) // Devuelve un objeto de jsonData. Este método arroja en el fracaso.
- NSJSONSerialization.dataWithJSONObject (jsonObject, opciones: NSJSONWritingOptions) // Devuelve NSData de un objeto JSON. Pase en NSJSONWritingOptions.PrettyPrinted en opciones para una salida que sea más legible.

Examples

Serialización, codificación y decodificación JSON con Apple Foundation y Swift Standard Library

La clase JSONSerialization está integrada en el marco de la Fundación de Apple.

2.2

Leer json

La función JSONObjectWithData toma NSData y devuelve AnyObject . Se puede usar as? para convertir el resultado a su tipo esperado.

```
do {
    guard let jsonData = "[\"Hello\", \"JSON\"]".dataUsingEncoding(NSUTF8StringEncoding) else
{
        fatalError("couldn't encode string as UTF-8")
    }

    // Convert JSON from NSData to AnyObject
    let jsonObject = try NSJSONSerialization.JSONObjectWithData(jsonData, options: [])

    // Try to convert AnyObject to array of strings
    if let stringArray = jsonObject as? [String] {
        print("Got array of strings: \(stringArray.joinWithSeparator(", "))")
    }
} catch {
    print("error reading JSON: \(error)")
}
```

Puede pasar las options: .AllowFragments que se options: .AllowFragments lugar de las options: [] para permitir la lectura de JSON cuando el objeto de nivel superior no es una matriz o diccionario.

Escribir json

Al llamar a dataWithJSONObject convierte un objeto compatible con JSON (matrices anidadas o diccionarios con cadenas, números y NSNull) en NSData bruto codificados como UTF-8.

```
do {
    // Convert object to JSON as NSData
    let jsonData = try NSJSONSerialization.dataWithJSONObject(jsonObject, options: [])
    print("JSON data: \(jsonData)")

    // Convert NSData to String
    let jsonString = String(data: jsonData, encoding: NSUTF8StringEncoding)!
    print("JSON string: \(jsonString)")
} catch {
    print("error writing JSON: \(error)")
}
```

Puede pasar las options: .PrettyPrinted lugar de las options: [] para impresión bonita.

3.0

Mismo comportamiento en Swift 3 pero con una sintaxis diferente.

```
do {
    guard let jsonData = "[\"Hello\", \"JSON\"]".data(using: String.Encoding.utf8) else {
        fatalError("couldn't encode string as UTF-8")
    }

    // Convert JSON from NSData to AnyObject
    let jsonObject = try JSONSerialization.jsonObject(with: jsonData, options: [])

    // Try to convert AnyObject to array of strings
    if let stringArray = jsonObject as? [String] {
        print("Got array of strings: \(stringArray.joined(separator: ", "))")
    }
} catch {
    print("error reading JSON: \(error)")
}

do {
    // Convert object to JSON as NSData
    let jsonData = try JSONSerialization.data(withJSONObject: jsonObject, options: [])
    print("JSON data: \(jsonData)")

    // Convert NSData to String
    let jsonString = String(data: jsonData, encoding: .utf8)!
    print("JSON string: \(jsonString)")
} catch {
    print("error writing JSON: \(error)")
}
```

> Nota: Lo siguiente está disponible actualmente solo en **Swift 4.0** y versiones posteriores.

A partir de Swift 4.0, la biblioteca estándar Swift incluye los protocolos Encodable y Decodable para definir un enfoque estandarizado para la codificación de datos y la decodificación. La adopción de estos protocolos permitirá que las implementaciones de los protocolos de Encoder y Decoder tomen sus datos y los codifiquen o descodifiquen hacia y desde una representación externa como JSON. La conformidad con la Codable protocolo combina tanto la Encodable y Decodable protocolos. Este es ahora el medio recomendado para manejar JSON en su programa.

Codificar y decodificar automáticamente

La forma más fácil de hacer un tipo codificable es declarar sus propiedades como tipos que ya son Codable . Estos tipos incluyen tipos de biblioteca estándar como String , Int y Double ; y tipos de fundaciones, como Date , Data y URL . Si las propiedades de un tipo son codificables, el tipo en sí se ajustará automáticamente a Codable simplemente declarando la conformidad.

Considere el siguiente ejemplo, en el que la estructura del Book se ajusta a Codable .

```
struct Book: Codable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

> Tenga en cuenta que las colecciones estándar como Array y Dictionary ajustan a Codable si contienen tipos codificables.

Al adoptar Codable , la estructura del Book ahora se puede codificar y descodificar desde JSON utilizando las clases JSONEncoder y JSONDecoder Apple Foundation, aunque Book no contiene ningún código para manejar específicamente JSON. Los codificadores y decodificadores personalizados también pueden escribirse, de conformidad con los protocolos Encoder y Decoder , respectivamente.

Codificar a datos JSON

```
// Create an instance of Book called book
let encoder = JSONEncoder()
let data = try! encoder.encode(book) // Do not use try! in production code
print(data)
```

> Configure encoder.outputFormatting = .prettyPrinted para una lectura más fácil. ## Decodificar a partir de datos JSON

Decodificar a partir de datos JSON

```
// Retrieve JSON string from some source
let jsonData = jsonString.data(encoding: .utf8)!
let decoder = JSONDecoder()
let book = try! decoder.decode(Book.self, for: jsonData) // Do not use try! in production code
print(book)
```

> En el ejemplo anterior, Book.self informa al decodificador del tipo al que se debe decodificar el JSON.

Codificación o decodificación exclusiva

A veces es posible que no necesite que los datos sean codificables y decodificables, como cuando solo necesita leer datos JSON de una API, o si su programa solo envía datos JSON a una API.

Si solo desea escribir datos JSON, ajuste su tipo a Encodable .

```
struct Book: Encodable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

Si solo desea leer datos JSON, ajuste su tipo a Decodable .

```
struct Book: Decodable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

Usando Nombres Clave Personalizados

Las API utilizan con frecuencia convenciones de nomenclatura distintas de la caja de camello estándar de Swift, como la caja de serpiente. Esto puede convertirse en un problema cuando se trata de decodificar JSON, ya que, de forma predeterminada, las claves JSON deben alinearse exactamente con los nombres de propiedad de su tipo. Para manejar estos escenarios, puede crear claves personalizadas para su tipo usando el protocolo CodingKey .

```
struct Book: Codable {
    // ...
    enum CodingKeys: String, CodingKey {
        case title
```

```
        case authors
        case publicationDate = "publication_date"
    }
}
```

CodingKeys se generan automáticamente para los tipos que adoptan el protocolo Codable , pero al crear nuestra propia implementación en el ejemplo anterior, permitimos que nuestro decodificador Codable coincidir la fecha de publicationDate caso de camello local con la fecha de publication_date caso de serpiente como lo entrega la API.

SwiftyJSON

SwiftyJSON es un marco Swift creado para eliminar la necesidad de un encadenamiento opcional en la serialización JSON normal.

Puede descargarlo aquí: https://github.com/SwiftyJSON/SwiftyJSON

Sin SwiftyJSON, su código se vería así para encontrar el nombre del primer libro en un objeto JSON:

```
if let jsonObject = try NSJSONSerialization.JSONObjectWithData(data, options: .AllowFragments)
as? [[String: AnyObject]],
let bookName = (jsonObject[0]["book"] as? [String: AnyObject])?["name"] as? String {
    //We can now use the book name
}
```

En SwiftyJSON, esto se simplifica enormemente:

```
let json = JSON(data: data)
if let bookName = json[0]["book"]["name"].string {
    //We can now use the book name
}
```

Elimina la necesidad de verificar todos los campos, ya que devolverá cero si alguno de ellos no es válido.

Para usar SwiftyJSON, descargue la versión correcta desde el repositorio de Git. Hay una rama para Swift 3. Simplemente arrastre el "SwiftyJSON.swift" a su proyecto e importe a su clase:

```
import SwiftyJSON
```

Puedes crear tu objeto JSON usando los siguientes dos inicializadores:

```
let jsonObject = JSON(data: dataObject)
```

o

```
let jsonObject = JSON(jsonObject) //This could be a string in a JSON format for example
```

Para acceder a sus datos, utilice subíndices:

```
let firstObjectInAnArray = jsonObject[0]
let nameOfFirstObject = jsonObject[0]["name"]
```

Luego, puede analizar su valor a un determinado tipo de datos, que devolverá un valor opcional:

```
let nameOfFirstObject = jsonObject[0]["name"].string //This will return the name as a string
```

```
let nameOfFirstObject = jsonObject[0]["name"].double //This will return null
```

También puede compilar sus rutas en una matriz rápida:

```
let convolutedPath = jsonObject[0]["name"][2]["lastName"]["firstLetter"].string
```

Es lo mismo que:

```
let convolutedPath = jsonObject[0, "name", 2, "lastName", "firstLetter"].string
```

SwiftyJSON también tiene funcionalidad para imprimir sus propios errores:

```
if let name = json[1337].string {
    //You can use the value - it is valid
} else {
    print(json[1337].error) // "Array[1337] is out of bounds" - You cant use the value
}
```

Si necesita escribir en su objeto JSON, puede usar subíndices de nuevo:

```
var originalJSON:JSON = ["name": "Jack", "age": 18]
originalJSON["age"] = 25 //This changes the age to 25
originalJSON["surname"] = "Smith" //This creates a new field called "surname" and adds the
value to it
```

Si necesita la cadena original para el JSON, por ejemplo, si necesita escribirla en un archivo, puede obtener el valor en bruto:

```
if let string = json.rawString() { //This is a String object
    //Write the string to a file if you like
}

if let data = json.rawData() { //This is an NSData object
    //Send the data to your server if you like
}
```

Freddy

Freddy es una biblioteca de análisis JSON mantenida por Big Nerd Ranch . Tiene tres ventajas principales:

1. Escriba seguridad: lo ayuda a trabajar con el envío y la recepción de JSON de forma que se eviten bloqueos en el tiempo de ejecución.

2. Idiomatic: aprovecha los genéricos, las enumeraciones y las funciones funcionales de Swift, sin documentación complicada ni operadores mágicos personalizados.

3. Manejo de errores: proporciona información de errores informativa para errores JSON comunes.

Ejemplo de datos JSON

Definamos algunos ejemplos de datos JSON para usar con estos ejemplos.

```
{
  "success": true,
  "people": [
```

```
    {
      "name": "Matt Mathias",
      "age": 32,
      "spouse": true
    },
    {
      "name": "Sergeant Pepper",
      "age": 25,
      "spouse": false
    }
  ],
  "jobs": [
    "teacher",
    "judge"
  ],
  "states": {
    "Georgia": [
      30301,
      30302,
      30303
    ],
    "Wisconsin": [
      53000,
      53001
    ]
  }
}
```

```
let jsonString = "{\"success\": true, \"people\": [{\"name\": \"Matt Mathias\",\"age\":
32,\"spouse\": true},{\"name\": \"Sergeant Pepper\",\"age\": 25,\"spouse\": false}],\"jobs\":
[\"teacher\",\"judge\"],\"states\": {\"Georgia\": [30301,30302,30303],\"Wisconsin\":
[53000,53001]}}"
let jsonData = jsonString.dataUsingEncoding(NSUTF8StringEncoding)!
```

Deserialización de datos sin procesar

Para deserializar los datos, inicializamos un objeto JSON luego accedemos a una clave particular.

```
do {
    let json = try JSON(data: jsonData)
    let success = try json.bool("success")
} catch {
    // do something with the error
}
```

Nosotros try aquí porque el acceso a la json para la tecla de "success" podría fallar - que podría no existe, o el valor para no ser un valor lógico.

También podemos especificar una ruta para acceder a los elementos anidados en la estructura JSON. La ruta es una lista de claves e índices separados por comas que describen la ruta a un valor de interés.

```
do {
    let json = try JSON(data: jsonData)
    let georgiaZipCodes = try json.array("states", "Georgia")
    let firstPersonName = try json.string("people", 0, "name")
} catch {
    // do something with the error
}
```

Deserializando modelos directamente

JSON se puede analizar directamente a una clase modelo que implementa el protocolo JSONDecodable .

```
public struct Person {
    public let name: String
    public let age: Int
    public let spouse: Bool
}

extension Person: JSONDecodable {
    public init(json: JSON) throws {
        name = try json.string("name")
        age = try json.int("age")
        spouse = try json.bool("spouse")
    }
}

do {
    let json = try JSON(data: jsonData)
    let people = try json.arrayOf("people", type: Person.self)
} catch {
    // do something with the error
}
```

Serialización de datos en bruto

Cualquier valor JSON puede ser serializado directamente a NSData .

```
let success = JSON.Bool(false)
let data: NSData = try success.serialize()
```

Serialización de modelos directamente

Cualquier clase de modelo que implemente el protocolo JSONEncodable se puede serializar directamente a NSData .

```
extension Person: JSONEncodable {
    public func toJSON() -> JSON {
        return .Dictionary([
            "name": .String(name),
            "age": .Int(age),
            "spouse": .Bool(spouse)
        ])
    }
}

let newPerson = Person(name: "Glenn", age: 23, spouse: true)
let data: NSData = try newPerson.toJSON().serialize()
```

Flecha

Arrow es una elegante biblioteca de análisis JSON en Swift.

Permite analizar JSON y asignarlo a clases de modelos personalizados con la ayuda de un operador <-- :

```
identifier <-- json["id"]
```

```
name <-- json["name"]
stats <-- json["stats"]
```

Ejemplo:

Modelo swift

```
struct Profile {
    var identifier = 0
    var name = ""
    var link: NSURL?
    var weekday: WeekDay = .Monday
    var stats = Stats()
    var phoneNumbers = [PhoneNumber]()
}
```

Archivo JSON

```
{
    "id": 15678,
    "name": "John Doe",
    "link": "https://apple.com/steve",
    "weekdayInt" : 3,
    "stats": {
        "numberOfFriends": 163,
        "numberOfFans": 10987
    },
    "phoneNumbers": [{
                     "label": "house",
                     "number": "9809876545"
                     }, {
                     "label": "cell",
                     "number": "0908070656"
                     }, {
                     "label": "work",
                     "number": "0916570656"
    }]
}
```

Cartografía

```
extension Profile: ArrowParsable {
    mutating func deserialize(json: JSON) {
        identifier <-- json["id"]
        link <-- json["link"]
        name <-- json["name"]
        weekday <-- json["weekdayInt"]
        stats <- json["stats"]
        phoneNumbers <-- json["phoneNumbers"]
    }
}
```

Uso

```
let profile = Profile()
profile.deserialize(json)
```

Instalación:

Cartago

```
github "s4cha/Arrow"
```

CocoaPods

```
pod 'Arrow'
use_frameworks!
```

A mano

Simplemente copie y pegue Arrow.swift en su proyecto Xcode

https://github.com/s4cha/Arrow

Como un marco

Descargue Arrow desde el repositorio de GitHub y genere el objetivo de Framework en el proyecto de ejemplo. Entonces enlace contra este marco.

JSON simple de análisis en objetos personalizados

Incluso si las bibliotecas de terceros son buenas, los protocolos proporcionan una forma sencilla de analizar JSON. Puedes imaginar que tienes un objeto Todo como

```
struct Todo {
    let comment: String
}
```

Cada vez que reciba el JSON, puede manejar el NSData simple como se muestra en el otro ejemplo utilizando el objeto NSJSONSerialization .

Después de eso, usando un protocolo simple JSONDecodable

```
typealias JSONDictionary = [String:AnyObject]
protocol JSONDecodable {
    associatedtype Element
    static func from(json json: JSONDictionary) -> Element?
}
```

Y hacer que tu estructura de Todo ajuste a JSONDecodable hace el truco

```
extension Todo: JSONDecodable {
    static func from(json json: JSONDictionary) -> Todo? {
        guard let comment = json["comment"] as? String else { return nil }
        return Todo(comment: comment)
    }
}
```

Puedes probarlo con este código json:

```
{
    "todos": [
        {
            "comment" : "The todo comment"
        }
    ]
}
```

Cuando lo obtuvo de la API, puede serializarlo como los ejemplos anteriores que se muestran en una instancia de AnyObject . Después de eso, puedes verificar si la instancia es una instancia de JSONDictionary

```
guard let jsonDictionary = dictionary as? JSONDictionary else { return }
```

La otra cosa a verificar, específica para este caso porque tiene una matriz de Todo en el JSON, es el diccionario todos

```
guard let todosDictionary = jsonDictionary["todos"] as? [JSONDictionary] else { return }
```

Ahora que tiene la matriz de diccionarios, puede convertir cada uno de ellos en un objeto Todo utilizando flatMap (eliminará automáticamente los valores nil de la matriz)

```
let todos: [Todo] = todosDictionary.flatMap { Todo.from(json: $0) }
```

JSON analizando Swift 3

Aquí está el archivo JSON que animals.json llamado animals.json

```
{
    "Sea Animals": [
    {
     "name": "Fish",
     "question": "How many species of fish are there?"    },
                {
                "name": "Sharks",
                "question": "How long do sharks live?"
                },
                {
                "name": "Squid",
                "question": "Do squids have brains?"
                },
                {
                "name": "Octopus",
                "question": "How big do octopus get?"
                },
                {
                "name": "Star Fish",
                "question": "How long do star fish live?"
                }
                ],
    "mammals": [
                {
      "name": "Dog",
      "question": "How long do dogs live?"
    },
    {
      "name": "Elephant",
      "question": "How much do baby elephants weigh?"
    },
    {
      "name": "Cats",
      "question": "Do cats really have 9 lives?"
    },
    {
      "name": "Tigers",
      "question": "Where do tigers live?"
    },
    {
      "name": "Pandas",
      "question": "WHat do pandas eat?"
    }   ] }
```

Importa tu archivo JSON a tu proyecto

Puede realizar esta sencilla función para imprimir su archivo JSON

```
    func jsonParsingMethod() {
         //get the file
         let filePath = Bundle.main.path(forResource: "animals", ofType: "json")
         let content = try! String(contentsOfFile: filePath!)

         let data: Data = content.data(using: String.Encoding.utf8)!
         let json: NSDictionary =  try! JSONSerialization.jsonObject(with: data as Data,
options:.mutableContainers) as! NSDictionary

//Call which part of the file you'd like to pare
         if let results = json["mammals"] as? [[String: AnyObject]] {

             for res in results {
                 //this will print out the names of the mammals from out file.
                 if let rates = res["name"] as? String {
                     print(rates)
                 }
            }
        }
     }
```

Si desea colocarlo en una vista de tabla, primero crearía un diccionario con un NSObject.

Cree un nuevo archivo swift llamado ParsingObject y cree sus variables de cadena.

> Asegúrese de que el nombre de la variable sea el mismo que el archivo JSON

. Por ejemplo, en nuestro proyecto tenemos name y question por lo tanto, en nuestro nuevo archivo swift, usaremos

```
var name: String?
var question: String?
```

Inicialice el objeto NSO que hicimos de nuevo en nuestro ViewController.swift var array = ParsingObject Luego, realizaríamos el mismo método que teníamos antes con una modificación menor.

```
      func jsonParsingMethod() {
             //get the file
             let filePath = Bundle.main.path(forResource: "animals", ofType: "json")
             let content = try! String(contentsOfFile: filePath!)

             let data: Data = content.data(using: String.Encoding.utf8)!
             let json: NSDictionary =  try! JSONSerialization.jsonObject(with: data as Data,
options:.mutableContainers) as! NSDictionary

//This time let's get Sea Animals
             let results = json["Sea Animals"] as? [[String: AnyObject]]

//Get all the stuff using a for-loop
             for i in 0 ..< results!.count {

//get the value
                 let dict = results?[i]
                 let resultsArray = ParsingObject()

//append the value to our NSObject file
                 resultsArray.setValuesForKeys(dict!)
```

```
                array.append(resultsArray)

            }

        }
```

Luego lo mostramos en nuestra vista de tabla haciendo esto,

```
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
            return array.count
        }

        func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
            let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath)
//This is where our values are stored
            let object = array[indexPath.row]
            cell.textLabel?.text = object.name
            cell.detailTextLabel?.text = object.question
            return cell
        }
```

Lea Lectura y escritura JSON en línea: https://riptutorial.com/es/swift/topic/223/lectura-y-escritura-json

Capítulo 39: Los diccionarios

Observaciones

Algunos ejemplos en este tema pueden tener un orden diferente cuando se usan porque el orden del diccionario no está garantizado.

Examples

Declarar Diccionarios

Los diccionarios son una colección desordenada de claves y valores. Los valores se relacionan con claves únicas y deben ser del mismo tipo.

Al inicializar un Diccionario, la sintaxis completa es la siguiente:

```
var books : Dictionary<Int, String> = Dictionary<Int, String>()
```

Aunque una forma más concisa de inicializar:

```
var books = [Int: String]()
// or
var books: [Int: String] = [:]
```

Declare un diccionario con claves y valores especificándolos en una lista separada por comas. Los tipos se pueden inferir de los tipos de claves y valores.

```
var books: [Int: String] = [1: "Book 1", 2: "Book 2"]
//books = [2: "Book 2", 1: "Book 1"]
var otherBooks = [3: "Book 3", 4: "Book 4"]
//otherBooks = [3: "Book 3", 4: "Book 4"]
```

Modificar los diccionarios

Añadir una clave y un valor a un diccionario

```
var books = [Int: String]()
//books = [:]
books[5] = "Book 5"
//books = [5: "Book 5"]
books.updateValue("Book 6", forKey: 5)
//[5: "Book 6"]
```

updateValue devuelve el valor original si existe o nulo.

```
let previousValue = books.updateValue("Book 7", forKey: 5)
//books = [5: "Book 7"]
//previousValue = "Book 6"
```

Eliminar valor y sus claves con sintaxis similar

```
books[5] = nil
//books [:]
books[6] = "Deleting from Dictionaries"
//books = [6: "Deleting from Dictionaries"]
let removedBook = books.removeValueForKey(6)
```

```
//books = [:]
//removedValue = "Deleting from Dictionaries"
```

Valores de acceso

Se puede acceder a un valor en un Dictionary usando su clave:

```
var books: [Int: String] = [1: "Book 1", 2: "Book 2"]
let bookName = books[1]
//bookName = "Book 1"
```

Los valores de un diccionario se pueden iterar mediante el uso de la propiedad de values :

```
for book in books.values {
    print("Book Title: \(book)")
}
//output: Book Title: Book 2
//output: Book Title: Book 1
```

De forma similar, las claves de un diccionario se pueden iterar mediante el uso de su propiedad keys :

```
for bookNumbers in books.keys {
    print("Book number: \(bookNumber)")
}
// outputs:
// Book number: 1
// Book number: 2
```

Para obtener todos key pares de key y value correspondientes entre sí (no obtendrá el orden correcto ya que es un Diccionario)

```
for (book,bookNumbers)in books{
print("\(book)  \(bookNumbers)")
}
// outputs:
// 2  Book 2
// 1  Book 1
```

Tenga en cuenta que un Dictionary , a diferencia de un Array , está intrínsecamente desordenado, es decir, no hay garantía sobre el pedido durante la iteración.

Si desea acceder a múltiples niveles de un Diccionario, use una sintaxis de subíndices repetida.

```
// Create a multilevel dictionary.
var myDictionary: [String:[Int:String]]! =
["Toys":[1:"Car",2:"Truck"],"Interests":[1:"Science",2:"Math"]]

print(myDictionary["Toys"][2]) // Outputs "Truck"
print(myDictionary["Interests"][1]) // Outputs "Science"
```

Cambiar el valor del diccionario usando la clave

```
var dict = ["name": "John", "surname": "Doe"]
// Set the element with key: 'name' to 'Jane'
dict["name"] = "Jane"
print(dict)
```

Obtener todas las claves en el diccionario

```
let myAllKeys = ["name" : "Kirit" , "surname" : "Modi"]
let allKeys = Array(myAllKeys.keys)
print(allKeys)
```

Fusionar dos diccionarios

```
extension Dictionary {
    func merge(dict: Dictionary<Key,Value>) -> Dictionary<Key,Value> {
        var mutableCopy = self
        for (key, value) in dict {
            // If both dictionaries have a value for same key, the value of the other
dictionary is used.
            mutableCopy[key] = value
        }
        return mutableCopy
    }
}
```

Lea Los diccionarios en línea: https://riptutorial.com/es/swift/topic/310/los-diccionarios

Capítulo 40: Manejo de errores

Observaciones

Para obtener más información sobre los errores, consulte El lenguaje de programación Swift .

Examples

Fundamentos de manejo de errores

Las funciones en Swift pueden devolver valores, **arrojar** errores o ambos:

```
func reticulateSplines()                // no return value and no error
func reticulateSplines() -> Int         // always returns a value
func reticulateSplines() throws         // no return value, but may throw an error
func reticulateSplines() throws -> Int  // may either return a value or throw an error
```

Cualquier valor que cumpla con el protocolo ErrorType (incluidos los objetos NSError) se puede lanzar como un error. Las enumeraciones proporcionan una forma conveniente de definir errores personalizados:

2.0 2.2

```
enum NetworkError: ErrorType {
    case Offline
    case ServerError(String)
}
```

3.0

```
enum NetworkError: Error {
    // Swift 3 dictates that enum cases should be `lowerCamelCase`
    case offline
    case serverError(String)
}
```

Un error indica una falla no fatal durante la ejecución del programa, y se maneja con las construcciones especializadas de flujo de control do / catch , throw , y try .

```
func fetchResource(resource: NSURL) throws -> String {
    if let (statusCode, responseString) = /* ...from elsewhere...*/ {
        if case 500..<600 = statusCode {
            throw NetworkError.serverError(responseString)
        } else {
            return responseString
        }
    } else {
        throw NetworkError.offline
    }
}
```

Los errores pueden ser atrapados con do / catch :

```
do {
    let response = try fetchResource(resURL)
    // If fetchResource() didn't throw an error, execution continues here:
    print("Got response: \(response)")
```

```
    ...
} catch {
    // If an error is thrown, we can handle it here.
    print("Whoops, couldn't fetch resource: \(error)")
}
```

Cualquier función que pueda lanzar un error ***debe*** ser llamada usando try , try? , o try! :

```
// error: call can throw but is not marked with 'try'
let response = fetchResource(resURL)

// "try" works within do/catch, or within another throwing function:
do {
    let response = try fetchResource(resURL)
} catch {
    // Handle the error
}

func foo() throws {
    // If an error is thrown, continue passing it up to the caller.
    let response = try fetchResource(resURL)
}

// "try?" wraps the function's return value in an Optional (nil if an error was thrown).
if let response = try? fetchResource(resURL) {
    // no error was thrown
}

// "try!" crashes the program at runtime if an error occurs.
let response = try! fetchResource(resURL)
```

Capturando diferentes tipos de error

Vamos a crear nuestro propio tipo de error para este ejemplo.

2.2

```
enum CustomError: ErrorType {
    case SomeError
    case AnotherError
}

func throwing() throws {
    throw CustomError.SomeError
}
```

3.0

```
enum CustomError: Error {
    case someError
    case anotherError
}

func throwing() throws {
    throw CustomError.someError
}
```

La sintaxis de Do-Catch permite capturar un error arrojado, y crea *automáticamente* un error nombre constante disponible en el bloque catch :

```
do {
    try throwing()
} catch {
    print(error)
}
```

También puedes declarar una variable tú mismo:

```
do {
    try throwing()
} catch let oops {
    print(oops)
}
```

También es posible encadenar diferentes declaraciones de catch . Esto es conveniente si se pueden lanzar varios tipos de errores en el bloque Do.

Aquí, el Do-Catch intentará primero lanzar el error como CustomError , luego como NSError si el tipo personalizado no coincide.

2.2

```
do {
    try somethingMayThrow()
} catch let custom as CustomError {
    print(custom)
} catch let error as NSError {
    print(error)
}
```

3.0

En Swift 3, no es necesario realizar una conversión explícita a NSError.

```
do {
    try somethingMayThrow()
} catch let custom as CustomError {
    print(custom)
} catch {
    print(error)
}
```

Patrón de captura y cambio para el manejo explícito de errores

```
class Plane {

    enum Emergency: ErrorType {
        case NoFuel
        case EngineFailure(reason: String)
        case DamagedWing
    }

    var fuelInKilograms: Int

    //... init and other methods not shown

    func fly() throws {
        // ...
        if fuelInKilograms <= 0 {
```

```
            // uh oh...
            throw Emergency.NoFuel
        }
    }

}
```

En la clase cliente:

```
let airforceOne = Plane()
do {
    try airforceOne.fly()
} catch let emergency as Plane.Emergency {
    switch emergency {
    case .NoFuel:
        // call nearest airport for emergency landing
    case .EngineFailure(let reason):
        print(reason) // let the mechanic know the reason
    case .DamagedWing:
        // Assess the damage and determine if the president can make it
    }
}
```

Deshabilitando la propagación de errores

Los creadores de Swift han puesto mucha atención en hacer que el lenguaje sea expresivo y el manejo de errores es exactamente eso, expresivo. Si intenta invocar una función que puede generar un error, la llamada a la función debe ir precedida por la palabra clave try. La palabra clave try no es mágica. Todo lo que hace es concienciar al desarrollador de la capacidad de lanzamiento de la función.

Por ejemplo, el siguiente código utiliza una función loadImage (atPath :), que carga el recurso de imagen en una ruta determinada o genera un error si la imagen no se puede cargar. En este caso, debido a que la imagen se envía con la aplicación, no se generará ningún error en el tiempo de ejecución, por lo que es apropiado deshabilitar la propagación de errores.

```
let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```

Crear error personalizado con descripción localizada

Crear enum de errores personalizados

```
enum RegistrationError: Error {
    case invalidEmail
    case invalidPassword
    case invalidPhoneNumber
}
```

Cree la extension de RegistrationError para manejar la descripción localizada.

```
extension RegistrationError: LocalizedError {
    public var errorDescription: String? {
        switch self {
        case .invalidEmail:
            return NSLocalizedString("Description of invalid email address", comment: "Invalid
Email")
        case .invalidPassword:
            return NSLocalizedString("Description of invalid password", comment: "Invalid
```

```
Password")
        case .invalidPhoneNumber:
            return NSLocalizedString("Description of invalid phoneNumber", comment: "Invalid
Phone Number")
        }
    }
}
```

Error de manejo:

```
let error: Error = RegistrationError.invalidEmail
print(error.localizedDescription)
```

Lea Manejo de errores en línea: https://riptutorial.com/es/swift/topic/283/manejo-de-errores

Capítulo 41: Marca de documentación

Examples

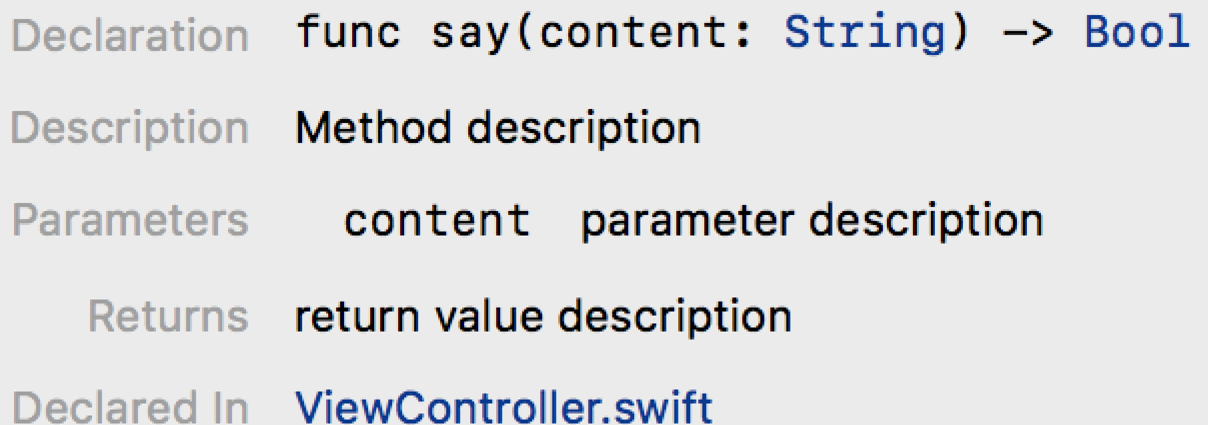
Documentación de la clase

Aquí hay un ejemplo básico de documentación de clase:

```
/// Class description
class Student {

    // Member description
    var name: String

    /// Method description
    ///
    /// - parameter content:   parameter description
    ///
    /// - returns: return value description
    func say(content: String) -> Bool {
        print("\(self.name) say \(content)")
        return true
    }
}
```

Tenga en cuenta que con *Xcode 8* , puede generar el fragmento de documentación con el comando + opción + / .

Esto volverá:



Estilos de documentación

```
/**
 Adds user to the list of poople which are assigned the tasks.

 - Parameter name: The name to add
 - Returns: A boolean value (true/false) to tell if user is added successfully to the people
list.
*/
func addMeToList(name: String) -> Bool {

    // Do something....
```

```
    return true
}
```

```
29
30     /**
31      Adds user to the list of people which are assigned the task
32
33      - Parameter name: The name to add
34      - Returns: A boolean value (true/false) to tell if user is a
35     */
36     func addMeToList(name: String) -> Bool {
```

Declaration func addMeToList(name: String) -> Bool
Description Adds user to the list of people which are assigned the tasks.
Parameters name The name to add
Returns A boolean value (true/false) to tell if user is added successfully to the people list.
Declared In ViewController.swift

```
44     /// This is a single line comment
```

```
/// This is a single line comment
func singleLineComment() {

}
```

```
44     /// This is a single line comment
45     func singleLineComment() {
```

Declaration func singleLineComment()
Description This is a single line comment
Declared In ViewController.swift

```
/**
 Repeats a string `times` times.

 - Parameter str:   The string to repeat.
 - Parameter times: The number of times to repeat `str`.

 - Throws: `MyError.InvalidTimes` if the `times` parameter
 is less than zero.

 - Returns: A new string with `str` repeated `times` times.
*/
func repeatString(str: String, times: Int) throws -> String {
    guard times >= 0 else { throw MyError.invalidTimes }
    return "Hello, world"
}
```

```
49
50      /**
51       Repeats a string `times` times.
52
53       - Parameter str:   The string to repeat.
54       - Parameter times: The number of times to repeat `str
55
56       - Throws: `MyError.InvalidTimes` if the `times` parame
57       is less than zero.
58
59       - Returns: A new string with `str` repeated `times` t
60       */
61      func repeatString(str: String, times: Int) throws -> S
                                                   yError.invalidTimes
```

Declaration func repeatString(str: String, times: Int) throws -> String

Description Repeats a string times times.

Parameters str The string to repeat.
times The number of times to repeat str.

Throws MyError.InvalidTimes if the times parameter is less than zero.

Returns A new string with str repeated times times.

Declared In ViewController.swift

```
/**
 # Lists

 You can apply *italic*, **bold**, or `code` inline styles.

 ## Unordered Lists
 - Lists are great,
 - but perhaps don't nest
 - Sub-list formatting
 - isn't the best.

 ## Ordered Lists
 1. Ordered lists, too
 2. for things that are sorted;
 3. Arabic numerals
 4. are the only kind supported.
*/
func complexDocumentation() {

}
```

```
70
71    /**
72     # Lists
73
74     You can apply *italic*, **bold**, or `code` inline
75
76     ## Unordered Lists
77     - Lists are great,
78     - but perhaps don't nest
79     - Sub-list formatting
80     - isn't the best.
81
82     ## Ordered Lists
83     1. Ordered lists, too
84     2. for things that are sorted;
85     3. Arabic numerals
86     4. are the only kind supported.
87     */
88    func complexDocumentation() {
```

Declaration `func complexDocumentation()`

Description

Lists

You can apply *italic*, **bold**, or code inline styles.

Unordered Lists

- Lists are great,
- but perhaps don't nest
- Sub-list formatting
- isn't the best.

Ordered Lists

1. Ordered lists, too
2. for things that are sorted;
3. Arabic numerals
4. are the only kind supported.

Declared In ViewController.swift

wn.
rtation.
d

```
/**
 Frame and construction style.

 - Road: For streets or trails.
 - Touring: For long journeys.
 - Cruiser: For casual trips around town.
 - Hybrid: For general-purpose transportation.
*/
enum Style {
    case Road, Touring, Cruiser, Hybrid
}
```

```
94    /**
95     Frame and construction style.
96
97     - Road: For streets or trails.
98     - Touring: For long journeys.
99     - Cruiser: For casual trips around town.
100    - Hybrid: For general-purpose transportation.
101    */
102   enum Style {
                                       Hybrid
```

Declaration enum Style

Description Frame and construction style.

- Road: For streets or trails.
- Touring: For long journeys.
- Cruiser: For casual trips around town.
- Hybrid: For general-purpose transportation.

Declared In ViewController.swift

```
111
```

Lea Marca de documentación en línea: https://riptutorial.com/es/swift/topic/6937/marca-de-documentacion

Capítulo 42: Método Swizzling

Observaciones

Cuando use Swizzling de métodos en Swift, hay dos requisitos que sus clases / métodos deben cumplir:

- Tu clase debe extender NSObject
- Las funciones que desea cambiar deben tener el atributo dynamic

Para obtener una explicación completa de por qué se requiere esto, consulte Uso de Swift con Cocoa y Objective-C :

> **Requerir el envío dinámico**
>
> Si bien el atributo @objc expone su Swift API al tiempo de ejecución de Objective-C, no garantiza el envío dinámico de una propiedad, método, subíndice o inicializador. *El compilador Swift aún puede desvirtualizar o acceder a los miembros en línea para optimizar el rendimiento de su código, sin pasar por el tiempo de ejecución de Objective-C* . Cuando marca una declaración de miembro con el modificador dynamic , el acceso a ese miembro siempre se despacha dinámicamente. Debido a que las declaraciones marcadas con el modificador dynamic se envían utilizando el tiempo de ejecución de Objective-C, se marcan implícitamente con el atributo @objc .
>
> Requerir el envío dinámico rara vez es necesario. **Sin embargo, debe usar el modificador dynamic cuando sepa que la implementación de una API se reemplaza en tiempo de ejecución** . Por ejemplo, puede usar la función method_exchangeImplementations en el tiempo de ejecución de Objective-C para intercambiar la implementación de un método mientras se ejecuta una aplicación. Si el compilador Swift incorporara la implementación del método o el acceso desvirtualizado a él, *la nueva implementación no se usaría* .

Campo de golf

Referencia en tiempo de ejecución de Objective-C

Método Swizzling en NSHipster

Examples

Ampliando UIViewController y Swizzling viewDidLoad

En Objective-C, el método swizzling es el proceso de cambiar la implementación de un selector existente. Esto es posible debido a la forma en que los selectores se asignan en una tabla de envío, o una tabla de punteros a funciones o métodos.

Los métodos Pure Swift no se distribuyen dinámicamente en el tiempo de ejecución de Objective-C, pero aún podemos aprovechar estos trucos en cualquier clase que hereda de NSObject .

Aquí, extenderemos UIViewController y swizzle viewDidLoad para agregar un registro personalizado:

```
extension UIViewController {

    // We cannot override load like we could in Objective-C, so override initialize instead
    public override static func initialize() {

        // Make a static struct for our dispatch token so only one exists in memory
        struct Static {
            static var token: dispatch_once_t = 0
```

```
        }

        // Wrap this in a dispatch_once block so it is only run once
        dispatch_once(&Static.token) {
            // Get the original selectors and method implementations, and swap them with our
new method
            let originalSelector = #selector(UIViewController.viewDidLoad)
            let swizzledSelector = #selector(UIViewController.myViewDidLoad)

            let originalMethod = class_getInstanceMethod(self, originalSelector)
            let swizzledMethod = class_getInstanceMethod(self, swizzledSelector)

            let didAddMethod = class_addMethod(self, originalSelector,
method_getImplementation(swizzledMethod), method_getTypeEncoding(swizzledMethod))

            // class_addMethod can fail if used incorrectly or with invalid pointers, so check
to make sure we were able to add the method to the lookup table successfully
            if didAddMethod {
                class_replaceMethod(self, swizzledSelector,
method_getImplementation(originalMethod), method_getTypeEncoding(originalMethod))
            } else {
                method_exchangeImplementations(originalMethod, swizzledMethod);
            }
        }
    }

    // Our new viewDidLoad function
    // In this example, we are just logging the name of the function, but this can be used to
run any custom code
    func myViewDidLoad() {
        // This is not recursive since we swapped the Selectors in initialize().
        // We cannot call super in an extension.
        self.myViewDidLoad()
        print(#function) // logs myViewDidLoad()
    }
}
```

Fundamentos de Swift Swiftling

Cambiemos la implementación de methodOne() y methodTwo() en nuestra clase TestSwizzling :

```
class TestSwizzling : NSObject {
    dynamic func methodOne()->Int{
        return 1
    }
}

extension TestSwizzling {

    //In Objective-C you'd perform the swizzling in load(),
    //but this method is not permitted in Swift
    override class func initialize()
    {

        struct Inner {
            static let i: () = {

                let originalSelector = #selector(TestSwizzling.methodOne)
                let swizzledSelector = #selector(TestSwizzling.methodTwo)
                let originalMethod = class_getInstanceMethod(TestSwizzling.self,
```

```
originalSelector);
                let swizzledMethod = class_getInstanceMethod(TestSwizzling.self,
swizzledSelector)
                method_exchangeImplementations(originalMethod, swizzledMethod)
            }
        }
        let _ = Inner.i
    }

    func methodTwo()->Int{
        // It will not be a recursive call anymore after the swizzling
        return methodTwo()+1
    }
}

var c = TestSwizzling()
print(c.methodOne())
print(c.methodTwo())
```

Fundamentos de Swizzling - Objective-C

Objective-C ejemplo de swizzling UIView's initWithFrame: method

```
static IMP original_initWithFrame;

+ (void)swizzleMethods {
    static BOOL swizzled = NO;
    if (!swizzled) {
        swizzled = YES;

        Method initWithFrameMethod =
            class_getInstanceMethod([UIView class], @selector(initWithFrame:));
        original_initWithFrame = method_setImplementation(
            initWithFrameMethod, (IMP)replacement_initWithFrame);
    }
}

static id replacement_initWithFrame(id self, SEL _cmd, CGRect rect) {

    // This will be called instead of the original initWithFrame method on UIView
    // Do here whatever you need...

    // Bonus: This is how you would call the original initWithFrame method
    UIView *view =
        ((id (*)(id, SEL, CGRect))original_initWithFrame)(self, _cmd, rect);

    return view;
}
```

Lea Método Swizzling en línea: https://riptutorial.com/es/swift/topic/1436/metodo-swizzling

Capítulo 43: NSRegularExpression en Swift

Observaciones

Caracteres especiales

```
*?+[(){}^$|\./
```

Examples

Extendiendo String para hacer patrones simples

```
extension String {
    func matchesPattern(pattern: String) -> Bool {
        do {
            let regex = try NSRegularExpression(pattern: pattern,
                                                options: NSRegularExpressionOptions(rawValue:
0))
            let range: NSRange = NSMakeRange(0, self.characters.count)
            let matches = regex.matchesInString(self, options: NSMatchingOptions(), range:
range)
            return matches.count > 0
        } catch _ {
            return false
        }
    }
}

// very basic examples - check for specific strings
dump("Pinkman".matchesPattern("(White|Pinkman|Goodman|Schrader|Fring)"))

// using character groups to check for similar-sounding impressionist painters
dump("Monet".matchesPattern("(M[oa]net)"))
dump("Manet".matchesPattern("(M[oa]net)"))
dump("Money".matchesPattern("(M[oa]net)"))     // false

// check surname is in list
dump("Skyler White".matchesPattern("\\w+ (White|Pinkman|Goodman|Schrader|Fring)"))

// check if string looks like a UK stock ticker
dump("VOD.L".matchesPattern("[A-Z]{2,3}\\.L"))
dump("BP.L".matchesPattern("[A-Z]{2,3}\\.L"))

// check entire string is printable ASCII characters
dump("tab\tformatted text".matchesPattern("^[\u{0020}-\u{007e}]*$"))

// Unicode example: check if string contains a playing card suit
dump("♠□".matchesPattern("[\u{2660}-\u{2667}]"))
dump("♡".matchesPattern("[\u{2660}-\u{2667}]"))
dump("□".matchesPattern("[\u{2660}-\u{2667}]"))    // false

// NOTE: regex needs Unicode-escaped characters
dump("♣□".matchesPattern("♣□"))           // does NOT work
```

A continuación se muestra otro ejemplo que se basa en lo anterior para hacer algo útil, que no se puede hacer fácilmente con ningún otro método y se presta bien a una solución de expresiones regulares.

```
// Pattern validation for a UK postcode.
// This simply checks that the format looks like a valid UK postcode and should not fail on
false positives.
private func isPostcodeValid(postcode: String) -> Bool {
    return postcode.matchesPattern("^[A-Z]{1,2}([0-9][A-Z]|[0-9]{1,2})\\s[0-9][A-Z]{2}")
}

// valid patterns (from
https://en.wikipedia.org/wiki/Postcodes_in_the_United_Kingdom#Validation)
// will return true
dump(isPostcodeValid("EC1A 1BB"))
dump(isPostcodeValid("W1A 0AX"))
dump(isPostcodeValid("M1 1AE"))
dump(isPostcodeValid("B33 8TH"))
dump(isPostcodeValid("CR2 6XH"))
dump(isPostcodeValid("DN55 1PT"))

// some invalid patterns
// will return false
dump(isPostcodeValid("EC12A 1BB"))
dump(isPostcodeValid("CRB1 6XH"))
dump(isPostcodeValid("CR 6XH"))
```

Uso básico

Hay varias consideraciones al implementar expresiones regulares en Swift.

```
let letters = "abcdefg"
let pattern = "[a,b,c]"
let regEx = try NSRegularExpression(pattern: pattern, options: [])
let nsString = letters as NSString
let matches = regEx.matches(in: letters, options: [], range: NSMakeRange(0, nsString.length))
let output = matches.map {nsString.substring(with: $0.range)}
//output = ["a", "b", "c"]
```

Para obtener una longitud de rango precisa que admita todos los tipos de caracteres, la cadena de entrada debe convertirse en una NSString.

Para la coincidencia de seguridad con un patrón se debe incluir un bloque de captura para manejar el fallo

```
let numbers = "121314"
let pattern = "1[2,3]"
do {
    let regEx = try NSRegularExpression(pattern: pattern, options: [])
    let nsString = numbers as NSString
    let matches = regEx.matches(in: numbers, options: [], range: NSMakeRange(0,
nsString.length))
    let output = matches.map {nsString.substring(with: $0.range)}
    output
} catch let error as NSError {
    print("Matching failed")
}
//output = ["12", "13"]
```

La funcionalidad de expresiones regulares a menudo se coloca en una extensión o ayudante para separar las preocupaciones.

Sustitución de subcadenas

Los patrones se pueden usar para reemplazar parte de una cadena de entrada.

El siguiente ejemplo reemplaza el símbolo de ciento con el símbolo de dólar.

```
var money = "¢¥€£$¥€£¢"
let pattern = "¢"
do {
    let regEx = try NSRegularExpression (pattern: pattern, options: [])
    let nsString = money as NSString
    let range = NSMakeRange(0, nsString.length)
    let correct$ = regEx.stringByReplacingMatches(in: money, options: .withTransparentBounds,
range: range, withTemplate: "$")
} catch let error as NSError {
    print("Matching failed")
}
//correct$ = "$¥€£$¥€£$"
```

Caracteres especiales

Para hacer coincidir los caracteres especiales, se debe usar la doble barra invertida `\.` becomes `\\.`

Los personajes de los que tendrás que escapar incluyen

```
(){}[]/\+*$>.|^?
```

El siguiente ejemplo obtiene tres tipos de soportes de apertura

```
let specials = "(){}[]"
let pattern = "(\\(|\\{|\\[)"
do {
    let regEx = try NSRegularExpression(pattern: pattern, options: [])
    let nsString = specials as NSString
    let matches = regEx.matches(in: specials, options: [], range: NSMakeRange(0,
nsString.length))
    let output = matches.map {nsString.substring(with: $0.range)}
} catch let error as NSError {
    print("Matching failed")
}
//output = ["(", "{", "["]
```

Validación

Las expresiones regulares se pueden usar para validar entradas al contar el número de coincidencias.

```
var validDate = false

let numbers = "35/12/2016"
let usPattern =  "^(0[1-9]|1[012])[-/.](0[1-9]|[12][0-9]|3[01])[-/.](19|20)\\d\\d$"
let ukPattern = "^(0[1-9]|[12][0-9]|3[01])[-/](0[1-9]|1[012])[-/](19|20)\\d\\d$"
do {
    let regEx = try NSRegularExpression(pattern: ukPattern, options: [])
    let nsString = numbers as NSString
    let matches = regEx.matches(in: numbers, options: [], range: NSMakeRange(0,
nsString.length))

    if matches.count > 0 {
```

```
        validDate = true
    }

    validDate

} catch let error as NSError {
    print("Matching failed")
}
//output = false
```

NSRegularExpression para validación de correo

```
func isValidEmail(email: String) -> Bool {

    let emailRegEx = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}"

    let emailTest = NSPredicate(format:"SELF MATCHES %@", emailRegEx)
    return emailTest.evaluate(with: email)
}
```

o podrías usar la extensión de cadena como esta:

```
extension String
{
    func isValidEmail() -> Bool {

        let emailRegEx = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}"

        let emailTest = NSPredicate(format:"SELF MATCHES %@", emailRegEx)
        return emailTest.evaluate(with: self)
    }
}
```

Lea NSRegularExpression en Swift en línea:
https://riptutorial.com/es/swift/topic/5763/nsregularexpression-en-swift

Capítulo 44: Números

Examples

Tipos de números y literales.

Los tipos numéricos incorporados de Swift son:

- Tamaño de palabra (dependiente de la arquitectura) firmado Int y sin firmar UInt .
- Enteros con signo de tamaño fijo Int8 , Int16 , Int32 , Int64 y enteros sin signo UInt8 , UInt16 , UInt32 , UInt64 .
- Tipos de punto flotante Float32 / Float , Float64 / Double y Float80 (solo x86).

Literales

El tipo de un literal numérico se deduce del contexto:

```
let x = 42    // x is Int by default
let y = 42.0  // y is Double by default

let z: UInt = 42     // z is UInt
let w: Float = -1    // w is Float
let q = 100 as Int8  // q is Int8
```

Los guiones bajos (_) se pueden usar para separar dígitos en literales numéricos. Se ignoran los ceros iniciales.

Los literales de punto flotante pueden especificarse usando partes de significando y exponente (*«significand»* **e** *«exponent»* para decimal; **0x** *«significand»* **p** *«exponent»* para hexadecimal.

Sintaxis literal entera

```
let decimal = 10               // ten
let decimal = -1000            // negative one thousand
let decimal = -1_000           // equivalent to -1000
let decimal = 42_42_42         // equivalent to 424242
let decimal = 0755             // equivalent to 755, NOT 493 as in some other languages
let decimal = 0123456789

let hexadecimal = 0x10         // equivalent to 16
let hexadecimal = 0x7FFFFFFF
let hexadecimal = 0xBadFace
let hexadecimal = 0x0123_4567_89ab_cdef

let octal = 0o10               // equivalent to 8
let octal = 0o755              // equivalent to 493
let octal = -0o0123_4567

let binary = -0b101010         // equivalent to -42
let binary = 0b111_101_101     // equivalent to 0o755
let binary = 0b1011_1010_1101  // equivalent to 0xB_A_D
```

Sintaxis literal de punto flotante

```
let decimal = 0.0
let decimal = -42.0123456789
let decimal = 1_000.234_567_89
```

```
let decimal = 4.567e5               // equivalent to 4.567×10⁵, or 456_700.0
let decimal = -2E-4                 // equivalent to -2×10⁻⁴, or -0.0002
let decimal = 1e+0                  // equivalent to 1×10⁰, or 1.0

let hexadecimal = 0x1p0             // equivalent to 1×2⁰, or 1.0
let hexadecimal = 0x1p-2            // equivalent to 1×2⁻², or 0.25
let hexadecimal = 0xFEEDp+3         // equivalent to 65261×2³, or 522088.0
let hexadecimal = 0x1234.5P4        // equivalent to 0x12345, or 74565.0
let hexadecimal = 0x123.45P8        // equivalent to 0x12345, or 74565.0
let hexadecimal = 0x12.345P12       // equivalent to 0x12345, or 74565.0
let hexadecimal = 0x1.2345P16       // equivalent to 0x12345, or 74565.0
let hexadecimal = 0x0.12345P20      // equivalent to 0x12345, or 74565.0
```

Convertir un tipo numérico a otro

```
func doSomething1(value: Double) { /* ... */ }
func doSomething2(value: UInt) { /* ... */ }

let x = 42               // x is an Int
doSomething1(Double(x))  // convert x to a Double
doSomething2(UInt(x))    // convert x to a UInt
```

Los inicializadores de enteros producen un **error de tiempo de ejecución** si el valor se desborda o se desborda:

```
Int8(-129.0) // fatal error: floating point value cannot be converted to Int8 because it is
less than Int8.min
Int8(-129)   // crash: EXC_BAD_INSTRUCTION / SIGILL
Int8(-128)   // ok
Int8(-2)     // ok
Int8(17)     // ok
Int8(127)    // ok
Int8(128)    // crash: EXC_BAD_INSTRUCTION / SIGILL
Int8(128.0)  // fatal error: floating point value cannot be converted to Int8 because it is
greater than Int8.max
```

La conversión de flotador a entero **redondea los valores hacia cero** :

```
Int(-2.2)  // -2
Int(-1.9)  // -1
Int(-0.1)  //  0
Int(1.0)   //  1
Int(1.2)   //  1
Int(1.9)   //  1
Int(2.0)   //  2
```

La conversión de entero a flotador puede tener **pérdidas** :

```
Int(Float(1_000_000_000_000_000_000))  // 999999984306749440
```

Convertir números a / desde cadenas

Use inicializadores de cadena para convertir números en cadenas:

```
String(1635999)                            // returns "1635999"
String(1635999, radix: 10)                 // returns "1635999"
String(1635999, radix: 2)                  // returns "110001111011010011111"
```

```
String(1635999, radix: 16)                    // returns "18f69f"
String(1635999, radix: 16, uppercase: true)   // returns "18F69F"
String(1635999, radix: 17)                    // returns "129gf4"
String(1635999, radix: 36)                    // returns "z2cf"
```

O use la interpolación de cadenas para casos simples:

```
let x = 42, y = 9001
"Between \(x) and \(y)"  // equivalent to "Between 42 and 9001"
```

Use inicializadores de tipos numéricos para convertir cadenas en números:

```
if let num = Int("42") { /* ... */ }                // num is 42
if let num = Int("Z2cF") { /* ... */ }              // returns nil (not a number)
if let num = Int("z2cf", radix: 36) { /* ... */ }   // num is 1635999
if let num = Int("Z2cF", radix: 36) { /* ... */ }   // num is 1635999
if let num = Int8("Z2cF", radix: 36) { /* ... */ }  // returns nil (too large for Int8)
```

Redondeo

redondo

Redondea el valor al número entero más cercano con x.5 redondeando hacia arriba (pero observa que -x.5 redondea hacia abajo).

```
round(3.000) // 3
round(3.001) // 3
round(3.499) // 3
round(3.500) // 4
round(3.999) // 4

round(-3.000) // -3
round(-3.001) // -3
round(-3.499) // -3
round(-3.500) // -4  *** careful here ***
round(-3.999) // -4
```

hacer techo

Redondea cualquier número con un valor decimal hasta el siguiente número entero más grande.

```
ceil(3.000) // 3
ceil(3.001) // 4
ceil(3.999) // 4

ceil(-3.000) // -3
ceil(-3.001) // -3
ceil(-3.999) // -3
```

piso

Redondea cualquier número con un valor decimal al siguiente número entero más pequeño.

```
floor(3.000) // 3
floor(3.001) // 3
floor(3.999) // 3
```

```
floor(-3.000) // -3
floor(-3.001) // -4
floor(-3.999) // -4
```

En t

Convierte un Double en un Int , eliminando cualquier valor decimal.

```
Int(3.000) // 3
Int(3.001) // 3
Int(3.999) // 3

Int(-3.000) // -3
Int(-3.001) // -3
Int(-3.999) // -3
```

Notas

- round , ceil y floor manejan arquitectura tanto de 64 como de 32 bits.

Generación de números aleatorios

arc4random_uniform(someNumber: UInt32) -> UInt32

Esto le da enteros aleatorios en el rango de 0 a someNumber - 1 .

El valor máximo para UInt32 es 4,294,967,295 (es decir, 2^32 - 1).

Ejemplos:

- Lanzamiento de moneda

  ```
  let flip = arc4random_uniform(2) // 0 or 1
  ```

- Tirada de dados

  ```
  let roll = arc4random_uniform(6) + 1 // 1...6
  ```

- Día aleatorio en octubre

  ```
  let day = arc4random_uniform(31) + 1 // 1...31
  ```

- Año aleatorio en los años 90.

  ```
  let year = 1990 + arc4random_uniform(10)
  ```

Forma general:

```
let number = min + arc4random_uniform(max - min + 1)
```

donde number , max y min son UInt32 .

Notas

- Hay un ligero sesgo de módulo con arc4random por arc4random_uniform se prefiere arc4random_uniform .

- Puede lanzar un valor UInt32 a un Int pero tenga cuidado de no estar fuera del rango.

Exposiciónción

En Swift, podemos **exponen- ciar** Double s con el método pow() incorporado:

```
pow(BASE, EXPONENT)
```

En el siguiente código, la base (5) se establece en la potencia del exponente (2):

```
let number = pow(5.0, 2.0) // Equals 25
```

Lea Números en línea: https://riptutorial.com/es/swift/topic/454/numeros

Capítulo 45: Objetos asociados

Examples

Propiedad, en una extensión de protocolo, lograda utilizando objeto asociado.

En Swift, las extensiones de protocolo no pueden tener propiedades verdaderas.

Sin embargo, en la práctica puede utilizar la técnica de "objeto asociado". El resultado es casi exactamente como una propiedad "real".

Aquí está la técnica exacta para agregar un "objeto asociado" a una extensión de protocolo:

Fundamentalmente, usas el objetivo-c "objc_getAssociatedObject" y las llamadas _set.

Las llamadas básicas son:

```
get {
   return objc_getAssociatedObject(self, & _Handle) as! YourType
   }
set {
   objc_setAssociatedObject(self, & _Handle, newValue, .OBJC_ASSOCIATION_RETAIN)
    }
```

Aquí hay un ejemplo completo. Los dos puntos críticos son:

1. En el protocolo, debe usar ": class" para evitar el problema de mutación.
2. En la extensión, debe usar "where Self: UIViewController" (o la clase apropiada) para dar el tipo de confirmación.

Entonces, para una propiedad de ejemplo "p":

```
import Foundation
import UIKit
import ObjectiveC          // don't forget this

var _Handle: UInt8 = 42    // it can be any value

protocol Able: class {
    var click:UIView? { get set }
    var x:CGFloat? { get set }
    // note that you >> do not << declare p here
}

extension Able where Self:UIViewController {

    var p:YourType { // YourType might be, say, an Enum
      get {
          return objc_getAssociatedObject(self, & _Handle) as! YourType
          // HOWEVER, SEE BELOW
          }
      set {
          objc_setAssociatedObject(self, & _Handle, newValue, .OBJC_ASSOCIATION_RETAIN)
          // often, you'll want to run some sort of "setter" here...
          __setter()
          }
    }

    func __setter() { something = p.blah() }
```

```
    func someOtherExtensionFunction() { p.blah() }
    // it's ok to use "p" inside other extension functions,
    // and you can use p anywhere in the conforming class
}
```

En cualquier clase conforme, ahora ha "agregado" la propiedad "p":

Puede usar "p" tal como usaría cualquier propiedad ordinaria en la clase conforme. Ejemplo:

```
class Clock:UIViewController, Able {
    var u:Int = 0

    func blah() {
      u = ...
      ... = u
      // use "p" as you would any normal property
      p = ...
      ... = p
    }

    override func viewDidLoad() {
      super.viewDidLoad()
      pm = .none // "p" MUST be "initialized" somewhere in Clock
    }
}
```

Nota. DEBES inicializar la pseudo-propiedad.

Xcode ***no* aplicará la** inicialización de "p" en la clase conforme.

Es esencial que inicialice "p", tal vez en viewDidLoad de la clase de confirmación.

Vale la pena recordar que p es en realidad una ***propiedad computada*** . En realidad, p es solo dos funciones, con azúcar sintáctico. No hay p "variable" en ninguna parte: el compilador no "asigna algo de memoria para p" en ningún sentido. Por esta razón, no tiene sentido esperar que Xcode aplique la "inicialización de p".

De hecho, para hablar con mayor precisión, debes recordar "usar p por primera vez, como si lo estuvieras inicializando". (De nuevo, eso es muy probable que esté en su código viewDidLoad).

Respecto al captador como tal.

Tenga en cuenta que se **bloqueará** si se llama al captador antes de establecer un valor para "p".

Para evitar eso, considere códigos tales como:

```
    get {
        let g = objc_getAssociatedObject(self, &_Handle)
        if (g == nil) {
            objc_setAssociatedObject(self, &_Handle, _default initial value_,
.OBJC_ASSOCIATION)
            return _default initial value_
        }
        return objc_getAssociatedObject(self, &_Handle) as! YourType
        }
```

Repetir. Xcode ***no* aplicará la** inicialización de p en la clase conforme. Es esencial que inicialice p, por ejemplo en viewDidLoad de la clase conforme.

Haciendo el código más simple ...

Es posible que desee utilizar estas dos funciones globales:

```
func _aoGet(_ ss: Any!, _ handlePointer: UnsafeRawPointer!, _ safeValue: Any!)->Any! {
    let g = objc_getAssociatedObject(ss, handlePointer)
    if (g == nil) {
        objc_setAssociatedObject(ss, handlePointer, safeValue, .OBJC_ASSOCIATION_RETAIN)
        return safeValue
    }
    return objc_getAssociatedObject(ss, handlePointer)
}

func _aoSet(_ ss: Any!, _ handlePointer: UnsafeRawPointer!, _ val: Any!) {
    objc_setAssociatedObject(ss, handlePointer, val, .OBJC_ASSOCIATION_RETAIN)
}
```

Tenga en cuenta que no hacen nada, excepto guardar la escritura y hacer que el código sea más legible. (Son esencialmente macros o funciones en línea).

Su código se convierte entonces en:

```
protocol PMable: class {
    var click:UILabel? { get set } // ordinary properties here
}

var _pHandle: UInt8 = 321

extension PMable where Self:UIViewController {

    var p:P {
        get {
            return _aoGet(self, &_pHandle, P() ) as! P
        }
        set {
            _aoSet(self, &_pHandle, newValue)
            __pmSetter()
        }
    }

    func __pmSetter() {
        click!.text = String(p)
    }

    func someFunction() {
        p.blah()
    }
}
```

(En el ejemplo de _aoGet, P es initalizable: en lugar de P (), puede usar "", 0 o cualquier valor predeterminado).

Lea Objetos asociados en línea: https://riptutorial.com/es/swift/topic/1085/objetos-asociados

Capítulo 46: Opcionales

Introducción

“Un valor opcional contiene un valor o contiene cero para indicar que falta un valor”

Extracto de: Apple Inc. "El lenguaje de programación Swift (Swift 3.1 Edition)." IBooks.
https://itun.es/us/k5SW7.l

Los casos de uso opcionales básicos incluyen: para una constante (let), el uso de un opcional dentro de un loop (if-let), desenvolver de forma segura un valor opcional dentro de un método (guarda-let), y como parte de los bucles de switch (case-let), por defecto a un valor si es nulo, usando el operador de fusión (??)

Sintaxis

- var optionalName: optionalType? // declara un tipo opcional, por defecto es nil
- var optionalName: optionalType? = valor // declara un opcional con un valor
- var optionalName: optionalType! // declara un opcional implícitamente sin envolver
- ¡Opcional! // forzar desenvolver un opcional

Observaciones

Para obtener más información sobre las opciones, consulte El lenguaje de programación Swift .

Examples

Tipos de opcionales

Los opcionales son un tipo de enumeración genérico que actúa como un contenedor. Esta envoltura permite que una variable tenga uno de dos estados: el valor del tipo definido por el usuario o nil , que representa la ausencia de un valor.

Esta habilidad es particularmente importante en Swift porque uno de los objetivos de diseño establecidos del lenguaje es trabajar bien con los marcos de Apple. Muchos (la mayoría) de los marcos de trabajo de Apple utilizan nil debido a su facilidad de uso y su importancia para los patrones de programación y el diseño de API en Objective-C.

En Swift, para que una variable tenga un valor nil , debe ser opcional. Se pueden crear opcionales agregando una ! o un ? al tipo de variable. Por ejemplo, para hacer un Int opcional, puedes usar

```
var numberOne: Int! = nil
var numberTwo: Int? = nil
```

? los opcionales deben ser explícitamente desenvueltos y deben usarse si no está seguro de si la variable tendrá un valor cuando acceda a ella. Por ejemplo, al convertir una cadena en un Int , el resultado es un Int? opcional Int? , porque se devolverá nil si la cadena no es un número válido

```
let str1 = "42"
let num1: Int? = Int(str1) // 42

let str2 = "Hello, World!"
let num2: Int? = Int(str2) // nil
```

! los opcionales se desenvuelven automáticamente y *solo* deben usarse cuando esté *seguro de* que la variable tendrá un valor cuando acceda a ella. Por ejemplo, un UIButton! global UIButton! variable que se inicializa en viewDidLoad()

```
//myButton will not be accessed until viewDidLoad is called,
//so a ! optional can be used here
var myButton: UIButton!

override func viewDidLoad(){
    self.myButton = UIButton(frame: self.view.frame)
    self.myButton.backgroundColor = UIColor.redColor()
    self.view.addSubview(self.myButton)
}
```

Desenvolviendo un opcional

Para poder acceder al valor de un opcional, se debe desempaquetar.

Condicionalmente, puede *desenvolver* un Opcional usando el enlace opcional y *forzar el desenvolver* un Opcional usando el ! operador.

El desenvolver condicionalmente pregunta "¿Tiene esta variable un valor?" mientras que la fuerza de desenvolvimiento dice "Esta variable tiene un valor".

Si obliga a desenvolver una variable que es nil , su programa arrojará un *nulo encontrado inesperadamente mientras desenvuelve una* excepción y falla *opcionales* , por lo que debe considerar cuidadosamente si lo usa ! es apropiado.

```
var text: String? = nil
var unwrapped: String = text! //crashes with "unexpectedly found nil while unwrapping an
Optional value"
```

Para un desenvolvimiento seguro, puede usar una sentencia if-let , que no arrojará una excepción o bloqueo si el valor envuelto es nil :

```
var number: Int?
if let unwrappedNumber = number {       // Has `number` been assigned a value?
    print("number: \(unwrappedNumber)") // Will not enter this line
} else {
    print("number was not assigned a value")
}
```

O, una declaración de guardia :

```
var number: Int?
guard let unwrappedNumber = number else {
    return
}
print("number: \(unwrappedNumber)")
```

Tenga en cuenta que el alcance de la variable unwrappedNumber está dentro de la sentencia if-let y fuera del bloque de guard .

Puede encadenar el desenvolvimiento de muchos opcionales, esto es principalmente útil en los casos en que su código requiere más de una variable para ejecutarse correctamente:

```
var firstName:String?
var lastName:String?

if let fn = firstName, let ln = lastName {
    print("\(fn) + \(ln)")//pay attention that the condition will be true only if both
optionals are not nil.
}
```

Tenga en cuenta que todas las variables deben ser desenvueltas para pasar la prueba con éxito, de lo contrario no tendría forma de determinar qué variables se desenvolvieron y cuáles no.

Puede encadenar declaraciones condicionales usando sus opcionales inmediatamente después de que se desenvuelva. Esto significa que no hay instrucciones anidadas si - si no!

```
var firstName:String? = "Bob"
var myBool:Bool? = false

if let fn = firstName, fn == "Bob", let bool = myBool, !bool {
    print("firstName is bob and myBool was false!")
}
```

Operador Coalescente Nil

Puede utilizar el *operador de unión nula* para desenvolver un valor si no es nulo, de lo contrario, proporcione un valor diferente:

```
func fallbackIfNil(str: String?) -> String {
    return str ?? "Fallback String"
}
print(fallbackIfNil("Hi")) // Prints "Hi"
print(fallbackIfNil(nil)) // Prints "Fallback String"
```

Este operador puede hacer un cortocircuito , lo que significa que si el operando de la izquierda no es nulo, el operando de la derecha no se evaluará:

```
func someExpensiveComputation() -> String { ... }

var foo : String? = "a string"
let str = foo ?? someExpensiveComputation()
```

En este ejemplo, como foo no es nulo, no se llamará a someExpensiveComputation() .

También puede encadenar varias declaraciones coalescentes nulas juntas:

```
var foo : String?
var bar : String?

let baz = foo ?? bar ?? "fallback string"
```

En este ejemplo, a baz se le asignará el valor no envuelto de foo si no es nulo, de lo contrario se le asignará el valor desenvuelto de la bar si no es nil, de lo contrario se le asignará el valor de reserva.

Encadenamiento opcional

Puede usar el encadenamiento opcional para llamar a un método , acceder a una propiedad o subíndice como opcional. Esto se hace colocando un ? entre la variable opcional dada y el miembro dado (método, propiedad o subíndice).

```
struct Foo {
    func doSomething() {
        print("Hello World!")
    }
}

var foo : Foo? = Foo()
```

```
foo?.doSomething() // prints "Hello World!" as foo is non-nil
```

Si foo contiene un valor, se doSomething() . Si foo es nil , entonces no ocurrirá nada malo: el código simplemente fallará en silencio y continuará ejecutándose.

```
var foo : Foo? = nil

foo?.doSomething() // will not be called as foo is nil
```

(Este es un comportamiento similar al envío de mensajes a nil en Objective-C)

La razón por la que el encadenamiento opcional se nombra como tal es porque la "opcionalidad" se propagará a través de los miembros a los que llama / accede. Lo que esto significa es que los valores de retorno de cualquier miembro utilizado con el encadenamiento opcional serán opcionales, independientemente de si están escritos como opcionales o no.

```
struct Foo {
    var bar : Int
    func doSomething() { ... }
}

let foo : Foo? = Foo(bar: 5)
print(foo?.bar) // Optional(5)
```

Aquí foo?.bar está devolviendo un Int? aunque la bar no es opcional, como foo sí es opcional.

A medida que se propaga la opcionalidad, los métodos que devuelven Void se devolverán Void? cuando se llama con el encadenamiento opcional. Esto puede ser útil para determinar si el método fue llamado o no (y por lo tanto si el opcional tiene un valor).

```
let foo : Foo? = Foo()

if foo?.doSomething() != nil {
    print("foo is non-nil, and doSomething() was called")
} else {
    print("foo is nil, therefore doSomething() wasn't called")
}
```

Aquí estamos comparando el Void? devuelva el valor con nil para determinar si se llamó al método (y, por lo tanto, si foo no es nulo).

Descripción general - ¿Por qué Optionals?

A menudo, cuando se programa, es necesario hacer una distinción entre una variable que tiene un valor y otra que no. Para tipos de referencia, como los punteros C, se puede usar un valor especial como null para indicar que la variable no tiene valor. Para los tipos intrínsecos, como un entero, es más difícil. Se puede usar un valor nominado, como -1, pero esto se basa en la interpretación del valor. También elimina ese valor "especial" del uso normal.

Para solucionar esto, Swift permite que cualquier variable se declare como opcional. Esto está indicado por el uso de un? o! después del tipo (ver Tipos de opcionales)

Por ejemplo,

```
var possiblyInt: Int?
```

declara una variable que puede o no contener un valor entero.

El valor especial nil indica que actualmente no hay ningún valor asignado a esta variable.

```
possiblyInt = 5      // PossiblyInt is now 5
possiblyInt = nil    // PossiblyInt is now unassigned
```

nil también se puede utilizar para probar un valor asignado:

```
if possiblyInt != nil {
    print("possiblyInt has the value \(possiblyInt!)")
}
```

Tenga en cuenta el uso de ! en la declaración de impresión para *desenvolver* el valor opcional.

Como ejemplo de un uso común de los opcionales, considere una función que devuelve un entero de una cadena que contiene dígitos; Es posible que la cadena contenga caracteres sin dígitos, o incluso que esté vacía.

¿Cómo puede una función que devuelve un simple Int indicar un error? No puede hacerlo devolviendo un valor específico ya que esto evitaría que ese valor se analice desde la cadena.

```
var someInt
someInt = parseInt("not an integer") // How would this function indicate failure?
```

En Swift, sin embargo, esa función puede simplemente devolver un Int *opcional* . Entonces el fallo es indicado por el valor de retorno de nil .

```
var someInt?
someInt = parseInt("not an integer")  // This function returns nil if parsing fails
if someInt == nil {
    print("That isn't a valid integer")
}
```

Lea Opcionales en línea: https://riptutorial.com/es/swift/topic/247/opcionales

Capítulo 47: Operadores Avanzados

Examples

Operadores personalizados

Swift soporta la creación de operadores personalizados. Los nuevos operadores se declaran a nivel global utilizando la palabra clave del operator .

La estructura del operador está definida por tres partes: ubicación de operandos, precedencia y asociatividad.

1. Los modificadores de prefix , infix y postfix se utilizan para iniciar una declaración de operador personalizada. Los modificadores prefix y postfix declaran si el operador debe ser antes o después, respectivamente, del valor sobre el que actúa. Dichos operadores son urnarios, como 8 y 3++ ** , ya que solo pueden actuar sobre un objetivo. El infix declara un operador binario, que actúa sobre los dos valores entre ellos, como 2+3 .

2. Los operadores con mayor **precedencia** se calculan primero. La precedencia del operador por defecto es solo mayor que ? ... : (un valor de 100 en Swift 2.x). La precedencia de los operadores Swift estándar se puede encontrar aquí .

3. **La asociatividad** define el orden de las operaciones entre operadores de la misma precedencia. Los operadores asociativos izquierdos se calculan de izquierda a derecha (orden de lectura, como la mayoría de los operadores), mientras que los operadores asociativos derechos calculan de derecha a izquierda.

3.0

A partir de Swift 3.0, uno definiría la precedencia y la asociatividad en un **grupo de precedencia en** lugar del operador mismo, de modo que múltiples operadores puedan compartir fácilmente la misma precedencia sin hacer referencia a los números crípticos. La lista de grupos de precedencia estándar se muestra a continuación .

Los operadores devuelven valores basados en el código de cálculo. Este código actúa como una función normal, con parámetros que especifican el tipo de entrada y la palabra clave de return especifica el valor calculado que el operador devuelve.

Aquí está la definición de un operador exponencial simple, ya que Swift estándar no tiene un operador exponencial.

```
import Foundation

infix operator ** { associativity left precedence 170 }

func ** (num: Double, power: Double) -> Double{
    return pow(num, power)
}
```

El infix dice que el operador ** trabaja entre dos valores, como 9**2 . Debido a que la función tiene asociatividad izquierda, 3**3**2 se calcula como (3**3)**2 . La prioridad de 170 es más alta que todas las operaciones Swift estándar, lo que significa que 3+2**4 calcula a 19 , a pesar de la asociatividad a la izquierda de ** .

3.0

```
import Foundation

infix operator **: BitwiseShiftPrecedence

func ** (num: Double, power: Double) -> Double {
```

```
    return pow(num, power)
}
```

En lugar de especificar explícitamente la precedencia y la asociatividad, en Swift 3.0 podríamos usar el grupo de precedencia integrado BitwiseShiftPrecedence que proporciona los valores correctos (igual que << , >>).

**: El incremento y la disminución están en desuso y se eliminarán en Swift 3.

Sobrecarga + para Diccionarios

Como actualmente no existe una forma sencilla de combinar diccionarios en Swift, puede ser útil sobrecargar los operadores + y += para agregar esta funcionalidad usando genéricos .

```
// Combines two dictionaries together. If both dictionaries contain
// the same key, the value of the right hand side dictionary is used.
func +<K, V>(lhs: [K : V], rhs: [K : V]) -> [K : V] {
    var combined = lhs
    for (key, value) in rhs {
        combined[key] = value
    }
    return combined
}

// The mutable variant of the + overload, allowing a dictionary
// to be appended to 'in-place'.
func +=<K, V>(inout lhs: [K : V], rhs: [K : V]) {
    for (key, value) in rhs {
        lhs[key] = value
    }
}
```

3.0

A partir de Swift 3, inout debe colocarse antes del tipo de argumento.

```
func +=<K, V>(lhs: inout [K : V], rhs: [K : V]) { ... }
```

Ejemplo de uso:

```
let firstDict = ["hello" : "world"]
let secondDict = ["world" : "hello"]
var thirdDict = firstDict + secondDict // ["hello": "world", "world": "hello"]

thirdDict += ["hello":"bar", "baz":"qux"] // ["hello": "bar", "baz": "qux", "world": "hello"]
```

Operadores conmutativos

Añadamos un operador personalizado para multiplicar un CGSize.

```
func *(lhs: CGFloat, rhs: CGSize) -> CGSize{
    let height = lhs*rhs.height
    let width = lhs*rhs.width
    return CGSize(width: width, height: height)
}
```

Ahora esto funciona

```
let sizeA = CGSize(height:100, width:200)
let sizeB = 1.1 * sizeA         //=> (height: 110, width: 220)
```

Pero si intentamos hacer la operación a la inversa, obtenemos un error.

```
let sizeC = sizeB * 20          // ERROR
```

Pero es lo suficientemente simple como para agregar:

```
func *(lhs: CGSize, rhs: CGFloat) -> CGSize{
    return rhs*lhs
}
```

Ahora el operador es conmutativo.

```
let sizeA = CGSize(height:100, width:200)
let sizeB = sizeA * 1.1              //=> (height: 110, width: 220)
```

Operadores de Bitwise

Los operadores de Swwise Bitwise le permiten realizar operaciones en la forma binaria de números. Puede especificar un literal binario prefijando el número con 0b , por lo que, por ejemplo, 0b110 es equivalente al número binario 110 (el número decimal 6). Cada 1 o 0 es un bit en el número.

Bitwise NO ~ :

```
var number: UInt8 = 0b01101100
let newNumber = ~number
// newNumber is equal to 0b01101100
```

Aquí, cada bit se cambia a su opuesto. Declarar el número como explícitamente UInt8 garantiza que el número sea positivo (para que no tengamos que lidiar con los negativos en el ejemplo) y que solo sea de 8 bits. Si 0b01101100 fuera un UInt más grande, habría 0 iniciales que se convertirían a 1 y serían significativos tras la inversión:

```
var number: UInt16 = 0b01101100
// number equals 0b0000000001101100
// the 0s are not significant
let newNumber = ~number
// newNumber equals 0b1111111110010011
// the 1s are now significant
```

- 0 -> 1
- 1 -> 0

Bitwise Y &

```
var number = 0b0110
let newNumber = number & 0b1010
// newNumber is equal to 0b0010
```

Aquí, un bit dado será 1 si y solo si los números binarios en ambos lados del operador & contienen un 1 en esa ubicación de bit.

- 0 y 0 -> 0
- 0 y 1 -> 0
- 1 y 1 -> 1

Bitwise o | :

```
var number = 0b0110
let newNumber = number | 0b1000
// newNumber is equal to 0b1110
```

Aquí, un bit dado será 1 si y solo si el número binario en al menos un lado de la | El operador contenía un 1 en esa ubicación de bit.

- 0 | 0 -> 0
- 0 | 1 -> 1
- 1 | 1 -> 1

Bitwise XOR (O exclusivo) ^ :

```
var number = 0b0110
let newNumber = number ^ 0b1010
// newNumber is equal to 0b1100
```

Aquí, un bit dado será 1 si y solo si los bits en esa posición de los dos operandos son diferentes.

- 0 ^ 0 -> 0
- 0 ^ 1 -> 1
- 1 ^ 1 -> 0

Para todas las operaciones binarias, el orden de los operandos no hace ninguna diferencia en el resultado.

Operadores de desbordamiento

Desbordamiento se refiere a lo que sucede cuando una operación resultaría en un número que es más grande o más pequeño que la cantidad designada de bits para que ese número pueda contener.

Debido a la forma en que funciona la aritmética binaria, después de que un número se vuelve demasiado grande para sus bits, el número se desborda hasta el número más pequeño posible (para el tamaño de bit) y luego continúa contando desde allí. De manera similar, cuando un número se vuelve demasiado pequeño, se desborda hasta el número más grande posible (por su tamaño de bit) y continúa contando desde allí.

Debido a que este comportamiento no es a menudo deseado y puede llevar a serios problemas de seguridad, los operadores aritméticos Swift + , - y * generarán errores cuando una operación cause un desbordamiento o subdesbordamiento. Para permitir explícitamente el desbordamiento y el desbordamiento, use &+ , &- , y &* lugar.

```
var almostTooLarge = Int.max
almostTooLarge + 1 // not allowed
almostTooLarge &+ 1 // allowed, but result will be the value of Int.min
```

Precedencia de los operadores Swift estándar.

Los operadores que se unen más estrechamente (mayor prioridad) se enumeran primero.

Los operadores	Grupo de precedencia (≥3.0)	Precedencia	Asociatividad
.		∞	izquierda
? , ! , ++ , -- , [] , () , {}	(sufijo)		

Los operadores	Grupo de precedencia (≥3.0)	Precedencia	Asociatividad
! , ~ , + , - , ++ , --	(prefijo)		
~> (swift ≤2.3)		255	izquierda
<< , >>	BitwiseShiftPrecedence	160	ninguna
* , / , % , & , &*	Multiplicación precedencia	150	izquierda
+ , - , | , ^ , &+ , &-	AdditionPrecedence	140	izquierda
... , ..<	RangeFormationPrecedence	135	ninguna
is as as? , as!	CastingPrecedence	132	izquierda
??	NilCoalescingPrecedence	131	Correcto
< , <= , > , >= , == != , === !== , ~=	Comparacion precedencia	130	ninguna
&&	Conexión lógicaPrincipio	120	izquierda
||	LogicDisjunctionPrecedence	110	izquierda
	DefaultPrecedence *		ninguna
? ... :	TernariaPrecedencia	100	Correcto
= , += , -= , *= , /= , %= , <<= , >>= , &= , |= , ^=	AsignacionPredecencia	90	derecho, asignación
->	FunctionArrowPrecedence		Correcto

3.0

- El grupo de precedencia de DefaultPrecedence es más alto que TernaryPrecedence , pero no está ordenado con el resto de los operadores. Aparte de este grupo, el resto de las precedentes son lineales.

- Esta tabla también se puede encontrar en la referencia API de Apple.
- La definición real de los grupos de precedencia se puede encontrar en el código fuente en GitHub

Lea Operadores Avanzados en línea: https://riptutorial.com/es/swift/topic/1048/operadores-avanzados

Capítulo 48: OptionSet

Examples

Protocolo OptionSet

OptionSetType es un protocolo diseñado para representar tipos de máscara de bits donde los bits individuales representan miembros del conjunto. Un conjunto de funciones lógicas y / o impone la sintaxis adecuada:

```
struct Features : OptionSet {
  let rawValue : Int
  static let none = Features(rawValue: 0)
  static let feature0 = Features(rawValue: 1 << 0)
  static let feature1 = Features(rawValue: 1 << 1)
  static let feature2 = Features(rawValue: 1 << 2)
  static let feature3 = Features(rawValue: 1 << 3)
  static let feature4 = Features(rawValue: 1 << 4)
  static let feature5 = Features(rawValue: 1 << 5)
  static let all: Features = [feature0, feature1, feature2, feature3, feature4, feature5]
}

Features.feature1.rawValue //2
Features.all.rawValue //63

var options: Features = [.feature1, .feature2, .feature3]

options.contains(.feature1) //true
options.contains(.feature4) //false

options.insert(.feature4)
options.contains(.feature4) //true


var otherOptions : Features = [.feature1, .feature5]

options.contains(.feature5) //false

options.formUnion(otherOptions)
options.contains(.feature5) //true

options.remove(.feature5)
options.contains(.feature5) //false
```

Lea OptionSet en línea: https://riptutorial.com/es/swift/topic/1242/optionset

Capítulo 49: Patrones de diseño - Creacionales

Introducción

Los patrones de diseño son soluciones generales a problemas que ocurren con frecuencia en el desarrollo de software. Las siguientes son plantillas de mejores prácticas estandarizadas para estructurar y diseñar códigos, así como ejemplos de contextos comunes en los que estos patrones de diseño serían apropiados.

Los patrones de diseño creacional abstraen la **creación de** instancias de objetos para hacer que un sistema sea más independiente del proceso de creación, composición y representación.

Examples

Semifallo

Los Singletons son un patrón de diseño de uso frecuente que consiste en una única instancia de una clase que se comparte a través de un programa.

En el siguiente ejemplo, creamos una propiedad static que contiene una instancia de la clase Foo . Recuerde que una propiedad static se comparte entre todos los objetos de una clase y no puede ser sobrescrita por subclases.

```
public class Foo
{
    static let shared = Foo()

    // Used for preventing the class from being instantiated directly
    private init() {}

    func doSomething()
    {
        print("Do something")
    }
}
```

Uso:

```
Foo.shared.doSomething()
```

Asegúrese de recordar el inicializador private :

> Esto asegura que sus singletons sean verdaderamente únicos e impide que los objetos externos creen sus propias instancias de su clase a través del control de acceso. Dado que todos los objetos vienen con un inicializador público predeterminado en Swift, debe anular su inicialización y hacerla privada. KrakenDev

Método de fábrica

> En la programación basada en clases, el patrón de método de fábrica es un patrón de creación que utiliza métodos de fábrica para tratar el problema de crear objetos sin tener que especificar la clase exacta del objeto que se creará. Referencia de Wikipedia

```
protocol SenderProtocol
{
    func send(package: AnyObject)
}
```

```
class Fedex: SenderProtocol
{
    func send(package: AnyObject)
    {
        print("Fedex deliver")
    }
}

class RegularPriorityMail: SenderProtocol
{
    func send(package: AnyObject)
    {
        print("Regular Priority Mail deliver")
    }
}

// This is our Factory
class DeliverFactory
{
    // It will be responsable for returning the proper instance that will handle the task
    static func makeSender(isLate isLate: Bool) -> SenderProtocol
    {
        return isLate ? Fedex() : RegularPriorityMail()
    }
}

// Usage:
let package = ["Item 1", "Item 2"]

// Fedex class will handle the delivery
DeliverFactory.makeSender(isLate:true).send(package)

// Regular Priority Mail class will handle the delivery
DeliverFactory.makeSender(isLate:false).send(package)
```

Al hacerlo, no dependemos de la implementación real de la clase, lo que hace que el sender() completamente transparente para quien lo consume.

En este caso, todo lo que necesitamos saber es que un remitente manejará la entrega y expondrá un método llamado send() . Hay varias otras ventajas: reducir el acoplamiento de clases, más fácil de probar, más fácil de agregar nuevos comportamientos sin tener que cambiar quién lo está consumiendo.

> Dentro del diseño orientado a objetos, las interfaces proporcionan capas de abstracción que facilitan la explicación conceptual del código y crean una barrera que impide las dependencias. Referencia de Wikipedia

Observador

> El patrón de observador es cuando un objeto, llamado sujeto, mantiene una lista de sus dependientes, llamados observadores, y les notifica automáticamente cualquier cambio de estado, generalmente llamando a uno de sus métodos. Se utiliza principalmente para implementar sistemas de manejo de eventos distribuidos. El patrón Observer es también una parte clave en el patrón arquitectónico familiar modelo-vista-controlador (MVC). Referencia de Wikipedia

Básicamente, el patrón de observador se usa cuando tienes un objeto que puede notificar a los observadores sobre ciertos comportamientos o cambios de estado.

Primero, creamos una referencia global (fuera de una clase) para el Centro de notificaciones:

```
let notifCentre = NotificationCenter.default
```

Genial ahora podemos llamar a esto desde cualquier lugar. Entonces querríamos registrar una clase como observador ...

```
notifCentre.addObserver(self, selector: #selector(self.myFunc), name: "myNotification",
object: nil)
```

Esto agrega la clase como un observador para "readForMyFunc". También indica que se debe llamar a la función myFunc cuando se recibe esa notificación. Esta función debe estar escrita en la misma clase:

```
func myFunc(){
    print("The notification has been received")
}
```

> Una de las ventajas de este patrón es que puede agregar muchas clases como observadores y, por lo tanto, realizar muchas acciones después de una notificación.

La notificación ahora puede simplemente enviarse (o publicarse si lo prefiere) desde casi cualquier lugar del código con la línea:

```
notifCentre.post(name: "myNotification", object: nil)
```

> También puede pasar información con la notificación como un Diccionario.

```
let myInfo = "pass this on"
notifCentre.post(name: "myNotification", object: ["moreInfo":myInfo])
```

Pero entonces necesitas agregar una notificación a tu función:

```
func myFunc(_ notification: Notification){
    let userInfo = (notification as NSNotification).userInfo as! [String: AnyObject]
    let passedInfo = userInfo["moreInfo"]
    print("The notification \(moreInfo) has been received")
    //prints - The notification pass this on has been received
}
```

Cadena de responsabilidad

> En el diseño orientado a objetos, el patrón de cadena de responsabilidad es un patrón de diseño que consiste en una fuente de objetos de command y una serie de objetos de processing . Cada objeto de processing contiene lógica que define los tipos de objetos de comando que puede manejar; el resto se pasa al siguiente objeto de processing en la cadena. También existe un mecanismo para agregar nuevos objetos de processing al final de esta cadena. Wikipedia

Configuración de las clases que conforman la cadena de responsabilidad.

Primero creamos una interfaz para todos los objetos de processing .

```
protocol PurchasePower {
var allowable : Float { get }
  var role : String { get }
  var successor : PurchasePower? { get set }
}

extension PurchasePower {
```

```
  func process(request : PurchaseRequest){
    if request.amount < self.allowable {
      print(self.role + " will approve $ \(request.amount) for \(request.purpose)")
    } else if successor != nil {
      successor?.process(request: request)
    }
  }
}
```

Luego creamos el objeto de command .

```
struct PurchaseRequest {
  var amount : Float
  var purpose : String
}
```

Finalmente, creando objetos que conforman la cadena de responsabilidad.

```
class ManagerPower : PurchasePower {
  var allowable: Float = 20
  var role : String = "Manager"
  var successor: PurchasePower?
}

class DirectorPower : PurchasePower {
  var allowable: Float = 100
  var role = "Director"
  var successor: PurchasePower?
}

class PresidentPower : PurchasePower {
  var allowable: Float = 5000
  var role = "President"
  var successor: PurchasePower?
}
```

Iniciar y encadenar juntos:

```
let manager = ManagerPower()
let director = DirectorPower()
let president = PresidentPower()

manager.successor = director
director.successor = president
```

El mecanismo para encadenar objetos aquí es el acceso a la propiedad.

Creando solicitud para ejecutarlo:

```
manager.process(request: PurchaseRequest(amount: 2, purpose: "buying a pen"))  // Manager will
approve $ 2.0 for buying a pen
manager.process(request: PurchaseRequest(amount: 90, purpose: "buying a printer")) // Director
will approve $ 90.0 for buying a printer

manager.process(request: PurchaseRequest(amount: 2000, purpose: "invest in stock")) //
President will approve $ 2000.0 for invest in stock
```

Iterador

> En la programación de computadoras, un iterador es un objeto que le permite a un programador atravesar un contenedor, particularmente listas. Wikipedia

```
struct Turtle {
  let name: String
}

struct Turtles {
  let turtles: [Turtle]
}

struct TurtlesIterator: IteratorProtocol {
  private var current = 0
  private let turtles: [Turtle]

  init(turtles: [Turtle]) {
    self.turtles = turtles
  }

  mutating func next() -> Turtle? {
    defer { current += 1 }
    return turtles.count > current ? turtles[current] : nil
  }
}

extension Turtles: Sequence {
  func makeIterator() -> TurtlesIterator {
    return TurtlesIterator(turtles: turtles)
  }
}
```

Y el ejemplo de uso sería

```
let ninjaTurtles = Turtles(turtles: [Turtle(name: "Leo"),
                                      Turtle(name: "Mickey"),
                                      Turtle(name: "Raph"),
                                      Turtle(name: "Doney")])
print("Splinter and")
for turtle in ninjaTurtles {
  print("The great: \(turtle)")
}
```

Patrón de constructor

> El patrón de construcción es un **patrón de diseño de software de creación de objetos** . A diferencia del patrón de fábrica abstracto y el patrón de método de fábrica cuya intención es habilitar el polimorfismo, la intención del patrón de construcción es encontrar una solución para el constructor telescópico anti-patrón. El antipatrón del constructor telescópico se produce cuando el aumento de la combinación de parámetros del constructor de objetos conduce a una lista exponencial de constructores. En lugar de utilizar numerosos constructores, el patrón de construcción utiliza otro objeto, un generador, que recibe cada parámetro de inicialización paso a paso y luego devuelve el objeto construido resultante de una vez.
>
> *-Wikipedia*

El objetivo principal del patrón de creación es configurar una configuración predeterminada para un objeto desde su creación. Es un intermediario entre el objeto que se construirá y todos los demás objetos relacionados con su creación.

Ejemplo:

Para que quede más claro, echemos un vistazo a un ejemplo de *Car Builder* .

Tenga en cuenta que tenemos una clase de *automóvil que* contiene muchas opciones para crear un objeto, como:

- Color.
- Numero de asientos.
- Número de ruedas.
- Tipo.
- Tipo de engranaje.
- Motor.
- Disponibilidad de bolsas de aire.

```
import UIKit

enum CarType {
    case

    sportage,
    saloon
}

enum GearType {
    case

    manual,
    automatic
}

struct Motor {
    var id: String
    var name: String
    var model: String
    var numberOfCylinders: UInt8
}

class Car: CustomStringConvertible {
    var color: UIColor
    var numberOfSeats: UInt8
    var numberOfWheels: UInt8
    var type: CarType
    var gearType: GearType
    var motor: Motor
    var shouldHasAirbags: Bool

    var description: String {
        return "color: \(color)\nNumber of seats: \(numberOfSeats)\nNumber of Wheels:
\(numberOfWheels)\n Type: \(gearType)\nMotor: \(motor)\nAirbag Availability:
\(shouldHasAirbags)"
    }

    init(color: UIColor, numberOfSeats: UInt8, numberOfWheels: UInt8, type: CarType, gearType:
GearType, motor: Motor, shouldHasAirbags: Bool) {

        self.color = color
        self.numberOfSeats = numberOfSeats
        self.numberOfWheels = numberOfWheels
        self.type = type
```

```
        self.gearType = gearType
        self.motor = motor
        self.shouldHasAirbags = shouldHasAirbags

    }
}
```

Creando un objeto de coche:

```
let aCar = Car(color: UIColor.black,
               numberOfSeats: 4,
               numberOfWheels: 4,
               type: .saloon,
               gearType: .automatic,
               motor: Motor(id: "101", name: "Super Motor",
                            model: "c4", numberOfCylinders: 6),
               shouldHasAirbags: true)

print(aCar)

/* Printing
 color: UIExtendedGrayColorSpace 0 1
 Number of seats: 4
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "101", name: "Super Motor", model: "c4", numberOfCylinders: 6)
 Airbag Availability: true
*/
```

El **problema que** surge al crear un objeto de automóvil es que el automóvil requiere que se creen muchos datos de configuración.

Para aplicar el patrón del generador, los parámetros del inicializador deben tener valores predeterminados *que pueden modificarse si es necesario* .

Clase CarBuilder:

```
class CarBuilder {
    var color: UIColor = UIColor.black
    var numberOfSeats: UInt8 = 5
    var numberOfWheels: UInt8 = 4
    var type: CarType = .saloon
    var gearType: GearType = .automatic
    var motor: Motor = Motor(id: "111", name: "Default Motor",
                             model: "T9", numberOfCylinders: 4)
    var shouldHasAirbags: Bool = false

    func buildCar() -> Car {
        return Car(color: color, numberOfSeats: numberOfSeats, numberOfWheels: numberOfWheels,
type: type, gearType: gearType, motor: motor, shouldHasAirbags: shouldHasAirbags)
    }
}
```

La clase CarBuilder define propiedades que podrían cambiarse para editar los valores del objeto de automóvil creado.

Construyamos autos nuevos usando el CarBuilder :

```
var builder = CarBuilder()
// currently, the builder creates cars with default configuration.
```

```
let defaultCar = builder.buildCar()
//print(defaultCar.description)
/* prints
 color: UIExtendedGrayColorSpace 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: false
*/

builder.shouldHasAirbags = true
// now, the builder creates cars with default configuration,
// but with a small edit on making the airbags available

let safeCar = builder.buildCar()
print(safeCar.description)
/* prints
 color: UIExtendedGrayColorSpace 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: true
 */

builder.color = UIColor.purple
// now, the builder creates cars with default configuration
// with some extra features: the airbags are available and the color is purple

let femaleCar = builder.buildCar()
print(femaleCar)
/* prints
 color: UIExtendedSRGBColorSpace 0.5 0 0.5 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: true
*/
```

La **ventaja** de aplicar el patrón de creación es la facilidad de crear objetos que deben contener gran parte de las configuraciones mediante el establecimiento de valores predeterminados, así como la facilidad de cambiar estos valores predeterminados.

Llevarlo mas alla:

Como una buena práctica, todas las propiedades que necesitan valores predeterminados deben estar en un *protocolo separado* , que debe ser implementado por la clase en sí y su generador.

Regresando a nuestro ejemplo, vamos a crear un nuevo protocolo llamado CarBluePrint :

```
import UIKit

enum CarType {
    case

    sportage,
    saloon
}
```

```
enum GearType {
    case

    manual,
    automatic
}

struct Motor {
    var id: String
    var name: String
    var model: String
    var numberOfCylinders: UInt8
}

protocol CarBluePrint {
    var color: UIColor { get set }
    var numberOfSeats: UInt8 { get set }
    var numberOfWheels: UInt8 { get set }
    var type: CarType { get set }
    var gearType: GearType { get set }
    var motor: Motor { get set }
    var shouldHasAirbags: Bool { get set }
}

class Car: CustomStringConvertible, CarBluePrint {
    var color: UIColor
    var numberOfSeats: UInt8
    var numberOfWheels: UInt8
    var type: CarType
    var gearType: GearType
    var motor: Motor
    var shouldHasAirbags: Bool

    var description: String {
        return "color: \(color)\nNumber of seats: \(numberOfSeats)\nNumber of Wheels:
\(numberOfWheels)\n Type: \(gearType)\nMotor: \(motor)\nAirbag Availability:
\(shouldHasAirbags)"
    }

    init(color: UIColor, numberOfSeats: UInt8, numberOfWheels: UInt8, type: CarType, gearType:
GearType, motor: Motor, shouldHasAirbags: Bool) {

        self.color = color
        self.numberOfSeats = numberOfSeats
        self.numberOfWheels = numberOfWheels
        self.type = type
        self.gearType = gearType
        self.motor = motor
        self.shouldHasAirbags = shouldHasAirbags

    }
}

class CarBuilder: CarBluePrint {
    var color: UIColor = UIColor.black
    var numberOfSeats: UInt8 = 5
    var numberOfWheels: UInt8 = 4
    var type: CarType = .saloon
    var gearType: GearType = .automatic
    var motor: Motor = Motor(id: "111", name: "Default Motor",
                             model: "T9", numberOfCylinders: 4)
```

```
    var shouldHasAirbags: Bool = false

    func buildCar() -> Car {
        return Car(color: color, numberOfSeats: numberOfSeats, numberOfWheels: numberOfWheels,
type: type, gearType: gearType, motor: motor, shouldHasAirbags: shouldHasAirbags)
    }
}
```

La ventaja de declarar las propiedades que necesitan un valor predeterminado en un protocolo es la obligación de implementar cualquier nueva propiedad agregada; Cuando una clase se ajusta a un protocolo, tiene que declarar todas sus propiedades / métodos.

Tenga en cuenta que hay una nueva característica requerida que se debe agregar al plan de creación de un automóvil llamado "nombre de la batería":

```
protocol CarBluePrint {
    var color: UIColor { get set }
    var numberOfSeats: UInt8 { get set }
    var numberOfWheels: UInt8 { get set }
    var type: CarType { get set }
    var gearType: GearType { get set }
    var motor: Motor { get set }
    var shouldHasAirbags: Bool { get set }

    // adding the new property
    var batteryName: String { get set }
}
```

Después de agregar la nueva propiedad, tenga en cuenta que surgirán dos errores de tiempo de compilación y notificará que para CarBluePrint protocolo CarBluePrint necesario declarar la propiedad 'batteryName'. Eso garantiza que CarBuilder declarará y establecerá un valor predeterminado para la propiedad batteryName .

Después de añadir batteryName nueva propiedad a CarBluePrint protocolo, la aplicación de ambos Car y CarBuilder clases debe ser:

```
class Car: CustomStringConvertible, CarBluePrint {
    var color: UIColor
    var numberOfSeats: UInt8
    var numberOfWheels: UInt8
    var type: CarType
    var gearType: GearType
    var motor: Motor
    var shouldHasAirbags: Bool
    var batteryName: String

    var description: String {
        return "color: \(color)\nNumber of seats: \(numberOfSeats)\nNumber of Wheels:
\(numberOfWheels)\nType: \(gearType)\nMotor: \(motor)\nAirbag Availability:
\(shouldHasAirbags)\nBattery Name: \(batteryName)"
    }

    init(color: UIColor, numberOfSeats: UInt8, numberOfWheels: UInt8, type: CarType, gearType:
GearType, motor: Motor, shouldHasAirbags: Bool, batteryName: String) {

        self.color = color
        self.numberOfSeats = numberOfSeats
        self.numberOfWheels = numberOfWheels
        self.type = type
        self.gearType = gearType
        self.motor = motor
```

```
        self.shouldHasAirbags = shouldHasAirbags
        self.batteryName = batteryName
    }
}

class CarBuilder: CarBluePrint {
    var color: UIColor = UIColor.red
    var numberOfSeats: UInt8 = 5
    var numberOfWheels: UInt8 = 4
    var type: CarType = .saloon
    var gearType: GearType = .automatic
    var motor: Motor = Motor(id: "111", name: "Default Motor",
                             model: "T9", numberOfCylinders: 4)
    var shouldHasAirbags: Bool = false
    var batteryName: String = "Default Battery Name"

    func buildCar() -> Car {
        return Car(color: color, numberOfSeats: numberOfSeats, numberOfWheels: numberOfWheels,
type: type, gearType: gearType, motor: motor, shouldHasAirbags: shouldHasAirbags, batteryName:
batteryName)
    }
}
```

Nuevamente, construyamos autos nuevos usando el CarBuilder :

```
var builder = CarBuilder()

let defaultCar = builder.buildCar()
print(defaultCar)
/* prints
 color: UIExtendedSRGBColorSpace 1 0 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: false
 Battery Name: Default Battery Name
*/

builder.batteryName = "New Battery Name"

let editedBatteryCar = builder.buildCar()
print(editedBatteryCar)
/*
 color: UIExtendedSRGBColorSpace 1 0 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: false
 Battery Name: New Battery Name
 */
```

Lea Patrones de diseño - Creacionales en línea:
https://riptutorial.com/es/swift/topic/4941/patrones-de-diseno---creacionales

Capítulo 50: Patrones de diseño - Estructurales

Introducción

Los patrones de diseño son soluciones generales a problemas que ocurren con frecuencia en el desarrollo de software. Las siguientes son plantillas de mejores prácticas estandarizadas para estructurar y diseñar códigos, así como ejemplos de contextos comunes en los que estos patrones de diseño serían apropiados.

Los patrones de diseño estructural se centran en la composición de clases y objetos para crear interfaces y lograr una mayor funcionalidad.

Examples

Adaptador

Los adaptadores se utilizan para convertir la interfaz de una clase determinada, conocida como **Adaptee** , en otra interfaz, llamada **Destino** . Las operaciones en el destino son llamadas por un **cliente** , y esas operaciones son *adaptadas* por el adaptador y pasadas al Adaptee.

En Swift, los adaptadores a menudo pueden formarse mediante el uso de protocolos. En el siguiente ejemplo, un Cliente capaz de comunicarse con el Destino cuenta con la capacidad de realizar funciones de la clase Adaptee mediante el uso de un adaptador.

```
// The functionality to which a Client has no direct access
class Adaptee {
    func foo() {
        // ...
    }
}

// Target's functionality, to which a Client does have direct access
protocol TargetFunctionality {
    func fooBar() {}
}

// Adapter used to pass the request on the Target to a request on the Adaptee
extension Adaptee: TargetFunctionality {
    func fooBar() {
        foo()
    }
}
```

Ejemplo de flujo de un adaptador unidireccional: Client -> Target -> Adapter -> Adaptee

Los adaptadores también pueden ser bidireccionales, y estos se conocen como **adaptadores de dos vías** . Un adaptador bidireccional puede ser útil cuando dos clientes diferentes necesitan ver un objeto de manera diferente.

Fachada

Una **fachada** proporciona una interfaz unificada de alto nivel para las interfaces del subsistema. Esto permite un acceso más sencillo y seguro a las instalaciones más generales de un subsistema.

El siguiente es un ejemplo de una fachada utilizada para establecer y recuperar objetos en UserDefaults.

```
enum Defaults {
```

```
    static func set(_ object: Any, forKey defaultName: String) {
        let defaults: UserDefaults = UserDefaults.standard
        defaults.set(object, forKey:defaultName)
        defaults.synchronize()
    }

    static func object(forKey key: String) -> AnyObject! {
        let defaults: UserDefaults = UserDefaults.standard
        return defaults.object(forKey: key) as AnyObject!
    }

}
```

El uso podría parecerse a lo siguiente.

```
Defaults.set("Beyond all recognition.", forKey:"fooBar")
Defaults.object(forKey: "fooBar")
```

Las complejidades de acceder a la instancia compartida y sincronizar los UserDefaults están ocultos para el cliente, y se puede acceder a esta interfaz desde cualquier parte del programa.

Lea Patrones de diseño - Estructurales en línea:
https://riptutorial.com/es/swift/topic/9497/patrones-de-diseno---estructurales

Capítulo 51: Programación Funcional en Swift

Examples

Extraer una lista de nombres de una lista de personas

Dada una estructura de Person

```
struct Person {
    let name: String
    let birthYear: Int?
}
```

y una matriz de Person(s)

```
let persons = [
    Person(name: "Walter White", birthYear: 1959),
    Person(name: "Jesse Pinkman", birthYear: 1984),
    Person(name: "Skyler White", birthYear: 1970),
    Person(name: "Saul Goodman", birthYear: nil)
]
```

podemos recuperar una matriz de String contiene la propiedad de name de cada Persona.

```
let names = persons.map { $0.name }
// ["Walter White", "Jesse Pinkman", "Skyler White", "Saul Goodman"]
```

Atravesando

```
let numbers = [3, 1, 4, 1, 5]
// non-functional
for (index, element) in numbers.enumerate() {
    print(index, element)
}

// functional
numbers.enumerate().map { (index, element) in
    print((index, element))
}
```

Saliente

La aplicación de una función a una colección / secuencia y la creación de una nueva colección / secuencia se llama **proyección** .

```
/// Projection
var newReleases = [
    [
        "id": 70111470,
        "title": "Die Hard",
        "boxart": "http://cdn-0.nflximg.com/images/2891/DieHard.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": [4.0],
        "bookmark": []
    ],
    [
```

```
        "id": 654356453,
        "title": "Bad Boys",
        "boxart": "http://cdn-0.nflximg.com/images/2891/BadBoys.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": [5.0],
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ],
    [
        "id": 65432445,
        "title": "The Chamber",
        "boxart": "http://cdn-0.nflximg.com/images/2891/TheChamber.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": [4.0],
        "bookmark": []
    ],
    [
        "id": 675465,
        "title": "Fracture",
        "boxart": "http://cdn-0.nflximg.com/images/2891/Fracture.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": [5.0],
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ]
]


var videoAndTitlePairs = [[String: AnyObject]]()
newReleases.map { e in
    videoAndTitlePairs.append(["id": e["id"] as! Int, "title": e["title"] as! String])
}

print(videoAndTitlePairs)
```

Filtración

Crear una secuencia seleccionando los elementos de una secuencia que pasan una determinada condición se llama **filtrado**

```
var newReleases = [
    [
        "id": 70111470,
        "title": "Die Hard",
        "boxart": "http://cdn-0.nflximg.com/images/2891/DieHard.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 4.0,
        "bookmark": []
    ],
    [
        "id": 654356453,
        "title": "Bad Boys",
        "boxart": "http://cdn-0.nflximg.com/images/2891/BadBoys.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 5.0,
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ],
    [
        "id": 65432445,
        "title": "The Chamber",
        "boxart": "http://cdn-0.nflximg.com/images/2891/TheChamber.jpg",
```

```
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 4.0,
        "bookmark": []
    ],
    [
        "id": 675465,
        "title": "Fracture",
        "boxart": "http://cdn-0.nflximg.com/images/2891/Fracture.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 5.0,
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ]
]

var videos1 = [[String: AnyObject]]()
/**
 *  Filtering using map
 */
newReleases.map { e in
    if e["rating"] as! Float == 5.0 {
        videos1.append(["id": e["id"] as! Int, "title": e["title"] as! String])
    }
}

print(videos1)

var videos2 = [[String: AnyObject]]()
/**
 *  Filtering using filter and chaining
 */
newReleases
    .filter{ e in
        e["rating"] as! Float == 5.0
    }
    .map { e in
    videos2.append(["id": e["id"] as! Int, "title": e["title"] as! String])
}

print(videos2)
```

Usando Filtro con Estructuras

Con frecuencia es posible que desee filtrar estructuras y otros tipos de datos complejos. Buscar en una matriz de estructuras para entradas que contienen un valor particular es una tarea muy común, y se logra fácilmente en Swift utilizando funciones de programación funcionales. Además, el código es extremadamente breve.

```
struct Painter {
    enum Type { case Impressionist, Expressionist, Surrealist, Abstract, Pop }
    var firstName: String
    var lastName: String
    var type: Type
}

let painters = [
    Painter(firstName: "Claude", lastName: "Monet", type: .Impressionist),
    Painter(firstName: "Edgar", lastName: "Degas", type: .Impressionist),
    Painter(firstName: "Egon", lastName: "Schiele", type: .Expressionist),
    Painter(firstName: "George", lastName: "Grosz", type: .Expressionist),
    Painter(firstName: "Mark", lastName: "Rothko", type: .Abstract),
```

```
    Painter(firstName: "Jackson", lastName: "Pollock", type: .Abstract),
    Painter(firstName: "Pablo", lastName: "Picasso", type: .Surrealist),
    Painter(firstName: "Andy", lastName: "Warhol", type: .Pop)
]

// list the expressionists
dump(painters.filter({$0.type == .Expressionist}))

// count the expressionists
dump(painters.filter({$0.type == .Expressionist}).count)
// prints "2"

// combine filter and map for more complex operations, for example listing all
// non-impressionist and non-expressionists by surname
dump(painters.filter({$0.type != .Impressionist && $0.type != .Expressionist})
    .map({$0.lastName}).joinWithSeparator(", "))
// prints "Rothko, Pollock, Picasso, Warhol"
```

Lea Programación Funcional en Swift en línea:
https://riptutorial.com/es/swift/topic/2948/programacion-funcional-en-swift

Capítulo 52: Protocolos

Introducción

Los protocolos son una forma de especificar cómo usar un objeto. Describen un conjunto de propiedades y métodos que una clase, estructura o enumeración deberían proporcionar, aunque los protocolos no imponen restricciones a la implementación.

Observaciones

Un protocolo Swift es una colección de requisitos que los tipos conformes deben implementar. El protocolo se puede usar en la mayoría de los lugares donde se espera un tipo, por ejemplo, matrices y requisitos genéricos.

Los miembros del protocolo siempre comparten el mismo calificador de acceso que todo el protocolo y no se pueden especificar por separado. Aunque un protocolo podría restringir el acceso con los requisitos de getter o setter, como se muestra en los ejemplos anteriores.

Para obtener más información sobre los protocolos, consulte El lenguaje de programación Swift .

Los protocolos Objective-C son similares a los protocolos Swift.

Los protocolos también son comparables a las interfaces de Java .

Examples

Conceptos básicos del protocolo

Acerca de los protocolos

Un protocolo especifica inicializadores, propiedades, funciones, subíndices y tipos asociados requeridos de un tipo de objeto Swift (clase, estructura o enumeración) conforme al protocolo. En algunos idiomas, ideas similares para especificaciones de requisitos de objetos subsiguientes se conocen como 'interfaces'.

Un protocolo declarado y definido es un tipo, en sí mismo, con una firma de sus requisitos establecidos, algo similar a la manera en que las funciones Swift son un tipo basado en su firma de parámetros y devoluciones.

Las especificaciones de Swift Protocol pueden ser opcionales, explícitamente requeridas y / o dadas implementaciones predeterminadas a través de una instalación conocida como Protocol Extensions. Un tipo de objeto Swift (clase, estructura o enumeración) que desee ajustarse a un Protocolo que se completa con Extensions para todos sus requisitos específicos solo debe indicar su deseo de cumplir con la conformidad total. La facilidad de implementación predeterminada de las Extensiones de Protocolo puede ser suficiente para cumplir con todas las obligaciones de conformidad con un Protocolo.

Los Protocolos pueden ser heredados por otros Protocolos. Esto, junto con las Extensiones de protocolo, significa que los Protocolos pueden y deben considerarse como una característica importante de Swift.

Los protocolos y extensiones son importantes para realizar los objetivos y enfoques más amplios de Swift para la flexibilidad del diseño de programas y los procesos de desarrollo. El propósito principal declarado de la capacidad de Protocolo y Extensión de Swift es facilitar el diseño compositivo en la arquitectura y el desarrollo del programa. Esto se conoce como Programación Orientada al Protocolo. Los viejos crustadores consideran esto superior a un enfoque en el diseño OOP.

Los protocolos definen interfaces que pueden ser implementadas por cualquier estructura , clase o enumeración :

```
protocol MyProtocol {
    init(value: Int)                      // required initializer
    func doSomething() -> Bool            // instance method
    var message: String { get }           // instance read-only property
    var value: Int { get set }            // read-write instance property
    subscript(index: Int) -> Int { get }  // instance subscript
    static func instructions() -> String  // static method
    static var max: Int { get }           // static read-only property
    static var total: Int { get set }     // read-write static property
}
```

Las propiedades definidas en los protocolos deben anotarse como `{ get }` o `{ get set }` . `{ get }` significa que la propiedad debe ser obtenible, y por lo tanto, se puede implementar como *cualquier* tipo de propiedad. `{ get set }` significa que la propiedad debe ser configurable así como también gettable.

Una estructura, clase o enumeración puede **ajustarse a** un protocolo:

```
struct MyStruct : MyProtocol {
    // Implement the protocol's requirements here
}
class MyClass : MyProtocol {
    // Implement the protocol's requirements here
}
enum MyEnum : MyProtocol {
    case caseA, caseB, caseC
    // Implement the protocol's requirements here
}
```

Un protocolo también puede definir una **implementación predeterminada** para cualquiera de sus requisitos a través de una extensión :

```
extension MyProtocol {

    // default implementation of doSomething() -> Bool
    // conforming types will use this implementation if they don't define their own
    func doSomething() -> Bool {
        print("do something!")
        return true
    }
}
```

Se puede **usar** un protocolo **como tipo** , siempre que no tenga requisitos de tipo associatedtype :

```
func doStuff(object: MyProtocol) {
    // All of MyProtocol's requirements are available on the object
    print(object.message)
    print(object.doSomething())
}

let items : [MyProtocol] = [MyStruct(), MyClass(), MyEnum.caseA]
```

También puede definir un tipo abstracto que se ajuste a **múltiples** protocolos:

3.0

Con Swift 3 o superior, esto se hace separando la lista de protocolos con un signo (`&`):

```
func doStuff(object: MyProtocol & AnotherProtocol) {
    // ...
```

```
}

let items : [MyProtocol & AnotherProtocol] = [MyStruct(), MyClass(), MyEnum.caseA]
```

3.0

Las versiones anteriores tienen un protocol<...> sintaxis protocol<...> donde los protocolos son una lista separada por comas entre los corchetes angulares <> .

```
protocol AnotherProtocol {
    func doSomethingElse()
}

func doStuff(object: protocol<MyProtocol, AnotherProtocol>) {

    // All of MyProtocol & AnotherProtocol's requirements are available on the object
    print(object.message)
    object.doSomethingElse()
}

// MyStruct, MyClass & MyEnum must now conform to both MyProtocol & AnotherProtocol
let items : [protocol<MyProtocol, AnotherProtocol>] = [MyStruct(), MyClass(), MyEnum.caseA]
```

Los tipos existentes se pueden **ampliar** para ajustarse a un protocolo:

```
extension String : MyProtocol {
    // Implement any requirements which String doesn't already satisfy
}
```

Requisitos de tipo asociado

Los protocolos pueden definir **los requisitos de tipo asociados** utilizando la palabra clave de **tipo** associatedtype :

```
protocol Container {
    associatedtype Element
    var count: Int { get }
    subscript(index: Int) -> Element { get set }
}
```

Los protocolos con requisitos de tipo asociados **solo pueden usarse como restricciones** genéricas :

```
// These are NOT allowed, because Container has associated type requirements:
func displayValues(container: Container) { ... }
class MyClass { let container: Container }
// > error: protocol 'Container' can only be used as a generic constraint
// > because it has Self or associated type requirements

// These are allowed:
func displayValues<T: Container>(container: T) { ... }
class MyClass<T: Container> { let container: T }
```

Un tipo que se ajusta al protocolo puede satisfacer un requisito de tipo associatedtype implícitamente, al proporcionar un tipo dado en el que el protocolo espera que aparezca el tipo associatedtype :

```
struct ContainerOfOne<T>: Container {
```

```
    let count = 1          // satisfy the count requirement
    var value: T

    // satisfy the subscript associatedtype requirement implicitly,
    // by defining the subscript assignment/return type as T
    // therefore Swift will infer that T == Element
    subscript(index: Int) -> T {
        get {
            precondition(index == 0)
            return value
        }
        set {
            precondition(index == 0)
            value = newValue
        }
    }
}

let container = ContainerOfOne(value: "Hello")
```

(Tenga en cuenta que para agregar claridad a este ejemplo, el tipo de marcador de posición genérico se denomina T ; un nombre más adecuado sería Element , lo que ocultaría el associatedtype Element del protocolo. El compilador todavía inferirá que el Element marcador de posición genérico se utiliza para satisfacer el tipo associatedtype Element Requisito del associatedtype Element .)

Un tipo associatedtype también puede satisfacerse explícitamente mediante el uso de typealias :

```
struct ContainerOfOne<T>: Container {

    typealias Element = T
    subscript(index: Int) -> Element { ... }

    // ...
}
```

Lo mismo ocurre con las extensiones:

```
// Expose an 8-bit integer as a collection of boolean values (one for each bit).
extension UInt8: Container {

    // as noted above, this typealias can be inferred
    typealias Element = Bool

    var count: Int { return 8 }
    subscript(index: Int) -> Bool {
        get {
            precondition(0 <= index && index < 8)
            return self & 1 << UInt8(index) != 0
        }
        set {
            precondition(0 <= index && index < 8)
            if newValue {
                self |= 1 << UInt8(index)
            } else {
                self &= ~(1 << UInt8(index))
            }
        }
    }
}
```

Si el tipo conforme ya cumple con el requisito, no se necesita implementación:

```
extension Array: Container {}  // Array satisfies all requirements, including Element
```

Patrón de delegado

Un *delegado* es un patrón de diseño común utilizado en los marcos de Cocoa y CocoaTouch, donde una clase delega la responsabilidad de implementar alguna funcionalidad a otra. Esto sigue un principio de separación de preocupaciones, donde la clase marco implementa una funcionalidad genérica mientras que una instancia delegada separada implementa el caso de uso específico.

Otra forma de ver el patrón de delegado es en términos de comunicación de objetos. Objects menudo necesitan conversar entre ellos y para hacerlo, un objeto debe ajustarse a un protocol para convertirse en un delegado de otro Objeto. Una vez que se ha realizado esta configuración, el otro objeto responde a sus delegados cuando suceden cosas interesantes.

Por ejemplo, una vista en la interfaz de usuario para mostrar una lista de datos debe ser responsable solo de la lógica de cómo se muestran los datos, no de decidir qué datos deben mostrarse.

Vayamos a un ejemplo más concreto. Si tienes dos clases, un padre y un niño:

```
class Parent { }
class Child { }
```

Y desea notificar a los padres de un cambio del niño.

En Swift, los delegados se implementan utilizando una declaración de protocol por lo que declararemos un protocol que implementará el delegate . Aquí el delegado es el objeto parent .

```
protocol ChildDelegate: class {
    func childDidSomething()
}
```

El niño debe declarar una propiedad para almacenar la referencia al delegado:

```
class Child {
    weak var delegate: ChildDelegate?
}
```

Observe la variable delegate es un opcional y el protocolo ChildDelegate está marcado sólo a ser implementado por tipo de clase (sin esto el delegate variable no puede ser declarada como una weak referencia evitando cualquier ciclo conservan. Esto significa que si el delegate la variable ya no es referenciado en cualquier otro lugar, será lanzado). Esto es para que la clase padre solo registre al delegado cuando sea necesario y esté disponible.

También para marcar a nuestro delegado como weak , debemos restringir nuestro protocolo ChildDelegate a tipos de referencia agregando palabras clave de class en la declaración de protocolo.

En este ejemplo, cuando el niño hace algo y necesita notificar a su padre, el niño llamará:

```
delegate?.childDidSomething()
```

Si el delegado ha sido definido, se le notificará al delegado que el niño ha hecho algo.

La clase principal deberá extender el protocolo ChildDelegate para poder responder a sus acciones. Esto se puede hacer directamente en la clase padre:

```
class Parent: ChildDelegate {
    ...
```

```
    func childDidSomething() {
        print("Yay!")
    }
}
```

O usando una extensión:

```
extension Parent: ChildDelegate {
    func childDidSomething() {
        print("Yay!")
    }
}
```

El padre también debe decirle al niño que es el delegado del niño:

```
// In the parent
let child = Child()
child.delegate = self
```

Por defecto, un protocol Swift no permite implementar una función opcional. Solo se pueden especificar si su protocolo está marcado con el atributo @objc y el modificador optional .

Por ejemplo, UITableView implementa el comportamiento genérico de una vista de tabla en iOS, pero el usuario debe implementar dos clases delegadas llamadas UITableViewDelegate y UITableViewDataSource que implementan el aspecto y el comportamiento de las células específicas.

```
@objc public protocol UITableViewDelegate : NSObjectProtocol, UIScrollViewDelegate {

        // Display customization
        optional public func tableView(tableView: UITableView, willDisplayCell cell:
UITableViewCell, forRowAtIndexPath indexPath: NSIndexPath)
        optional public func tableView(tableView: UITableView, willDisplayHeaderView view:
UIView, forSection section: Int)
        optional public func tableView(tableView: UITableView, willDisplayFooterView view:
UIView, forSection section: Int)
        optional public func tableView(tableView: UITableView, didEndDisplayingCell cell:
UITableViewCell, forRowAtIndexPath indexPath: NSIndexPath)
        ...
}
```

Puede implementar este protocolo cambiando su definición de clase, por ejemplo:

```
class MyViewController : UIViewController, UITableViewDelegate
```

Se debe implementar cualquier método que no esté marcado como optional en la definición de protocolo (UITableViewDelegate en este caso).

Ampliación del protocolo para una clase específica conforme

Puede escribir la **implementación del protocolo predeterminado** para una clase específica.

```
protocol MyProtocol {
    func doSomething()
}

extension MyProtocol where Self: UIViewController {
    func doSomething() {
        print("UIViewController default protocol implementation")
```

```
    }
}

class MyViewController: UIViewController, MyProtocol { }

let vc = MyViewController()
vc.doSomething() // Prints "UIViewController default protocol implementation"
```

Usando el protocolo RawRepresentable (Extensible Enum)

```
// RawRepresentable has an associatedType RawValue.
// For this struct, we will make the compiler infer the type
// by implementing the rawValue variable with a type of String
//
// Compiler infers RawValue = String without needing typealias
//
struct NotificationName: RawRepresentable {
    let rawValue: String

    static let dataFinished = NotificationNames(rawValue: "DataFinishedNotification")
}
```

Esta estructura puede extenderse a otro lugar para agregar casos

```
extension NotificationName {
    static let documentationLaunched = NotificationNames(rawValue:
"DocumentationLaunchedNotification")
}
```

Y una interfaz puede diseñar alrededor de cualquier tipo de RawRepresentable o específicamente su estructura de enumeración

```
func post(notification notification: NotificationName) -> Void {
    // use notification.rawValue
}
```

En el sitio de la llamada, puede usar la sintaxis de puntos para el NotificationName

```
post(notification: .dataFinished)
```

Usando la función genérica de RawRepresentable

```
// RawRepresentable has an associate type, so the
// method that wants to accept any type conforming to
// RawRepresentable needs to be generic
func observe<T: RawRepresentable>(object: T) -> Void {
    // object.rawValue
}
```

Protocolos de clase solamente

Un protocolo puede especificar que solo una clase puede implementarlo mediante el uso de la palabra clave de class en su lista de herencia. Esta palabra clave debe aparecer antes que cualquier otro protocolo heredado en esta lista.

```
protocol ClassOnlyProtocol: class, SomeOtherProtocol {
    // Protocol requirements
```

```
}
```

Si un tipo que no es de clase intenta implementar ClassOnlyProtocol , se ClassOnlyProtocol un error del compilador.

```
struct MyStruct: ClassOnlyProtocol {
    // error: Non-class type 'MyStruct' cannot conform to class protocol 'ClassOnlyProtocol'
}
```

Otros protocolos pueden heredar de ClassOnlyProtocol , pero tendrán el mismo requisito de clase solamente.

```
protocol MyProtocol: ClassOnlyProtocol {
    // ClassOnlyProtocol Requirements
    // MyProtocol Requirements
}

class MySecondClass: MyProtocol {
    // ClassOnlyProtocol Requirements
    // MyProtocol Requirements
}
```

Semántica de referencia de protocolos de clase solamente.

El uso de un protocolo solo de clase permite la semántica de referencia cuando el tipo de conformidad es desconocido.

```
protocol Foo : class {
    var bar : String { get set }
}

func takesAFoo(foo:Foo) {

    // this assignment requires reference semantics,
    // as foo is a let constant in this scope.
    foo.bar = "new value"
}
```

En este ejemplo, como Foo es un protocolo de solo clase, la asignación a la bar es válida ya que el compilador sabe que foo es un tipo de clase, y por lo tanto tiene una semántica de referencia.

Si Foo no fuera un protocolo solo de clase, se produciría un error de compilación, ya que el tipo conforme podría ser un tipo de valor , que requeriría una anotación de var para poder ser modificable.

```
protocol Foo {
    var bar : String { get set }
}

func takesAFoo(foo:Foo) {
    foo.bar = "new value" // error: Cannot assign to property: 'foo' is a 'let' constant
}
```

```
func takesAFoo(foo:Foo) {
    var foo = foo // mutable copy of foo
    foo.bar = "new value" // no error - satisfies both reference and value semantics
}
```

Variables débiles del tipo de protocolo.

Al aplicar el modificador weak a una variable de tipo de protocolo, ese tipo de protocolo debe ser solo de clase, ya que weak solo se puede aplicar a tipos de referencia.

```
weak var weakReference : ClassOnlyProtocol?
```

Implementando el protocolo Hashable

Los tipos utilizados en Sets y Dictionaries(key) deben cumplir con el protocolo Hashable que se hereda del protocolo Equatable .

Hashable debe implementar el tipo personalizado conforme al protocolo Hashable

- Una propiedad calculada hashValue
- Defina uno de los operadores de igualdad, es decir == o != .

El siguiente ejemplo implementa el protocolo Hashable para una struct personalizada:

```
struct Cell {
    var row: Int
    var col: Int

    init(_ row: Int, _ col: Int) {
        self.row = row
        self.col = col
    }
}

extension Cell: Hashable {

    // Satisfy Hashable requirement
    var hashValue: Int {
        get {
            return row.hashValue^col.hashValue
        }
    }

    // Satisfy Equatable requirement
    static func ==(lhs: Cell, rhs: Cell) -> Bool {
        return lhs.col == rhs.col && lhs.row == rhs.row
    }

}

// Now we can make Cell as key of dictonary
var dict = [Cell : String]()

dict[Cell(0, 0)] = "0, 0"
dict[Cell(1, 0)] = "1, 0"
dict[Cell(0, 1)] = "0, 1"

// Also we can create Set of Cells
var set = Set<Cell>()

set.insert(Cell(0, 0))
set.insert(Cell(1, 0))
```

Nota : No es necesario que los diferentes valores en el tipo personalizado tengan diferentes valores hash, las colisiones son aceptables. Si los valores de hash son iguales, el operador de

igualdad se utilizará para determinar la igualdad real.

Lea Protocolos en línea: https://riptutorial.com/es/swift/topic/241/protocolos

Capítulo 53: Reflexión

Sintaxis

- Espejo (reflejando: instancia) // Inicializa un espejo con el sujeto a reflejar
- mirror.displayStyle // Estilo de visualización utilizado para los parques infantiles Xcode
- mirror.description // Representación textual de esta instancia, vea CustomStringConvertible
- mirror.subjectType // Devuelve el tipo del tema que se refleja
- mirror.superclassMirror // Devuelve el espejo de la superclase del sujeto que se refleja

Observaciones

1. ***Observaciones generales:***

Un Mirror es una struct utilizada en la introspección de un objeto en Swift. Su propiedad más destacada es la matriz infantil. Un posible caso de uso es serializar una estructura para Core Data . Esto se hace al convertir una struct en un objeto NSManagedObject .

2. ***Uso básico para comentarios de espejo:***

El children propiedad de un Mirror es una matriz de objetos secundarios del objeto de la instancia espejo está reflejando. Un objeto child tiene dos propiedades label y value . Por ejemplo, un niño puede ser una propiedad con el nombre title y el valor de Game of Thrones: A Song of Ice and Fire .

Examples

Uso básico para espejo

Creando la clase para ser el sujeto del espejo.

```
class Project {
    var title: String = ""
    var id: Int = 0
    var platform: String = ""
    var version: Int = 0
    var info: String?
}
```

Creando una instancia que realmente será el sujeto del espejo. También aquí puede agregar valores a las propiedades de la clase Proyecto.

```
let sampleProject = Project()
sampleProject.title = "MirrorMirror"
sampleProject.id = 199
sampleProject.platform = "iOS"
sampleProject.version = 2
sampleProject.info = "test app for Reflection"
```

El siguiente código muestra la creación de la instancia de Mirror. La propiedad children del espejo es una AnyForwardCollection<Child> donde Child es typealias tuple para la propiedad y el valor del sujeto. Child tenía una label: String y value: Any .

```
let projectMirror = Mirror(reflecting: sampleProject)
let properties = projectMirror.children

print(properties.count)        //5
```

```
print(properties.first?.label) //Optional("title")
print(properties.first!.value) //MirrorMirror
print()

for property in properties {
    print("\(property.label!):\(property.value)")
}
```

Salida en Playground o Console en Xcode para el bucle for anterior.

```
title:MirrorMirror
id:199
platform:iOS
version:2
info:Optional("test app for Reflection")
```

Probado en Playground en Xcode 8 beta 2

Obtención de tipos y nombres de propiedades para una clase sin tener que instanciarla

El uso de la clase Swift Mirror funciona si desea extraer el *nombre* , el *valor* y el *tipo* (Swift 3: type(of: value) , Swift 2: value.dynamicType) de las propiedades para una **instancia** de una determinada clase.

Si la clase hereda de NSObject , puede usar el método class_copyPropertyList junto con property_getAttributes para averiguar el *nombre* y los *tipos* de propiedades de una clase, **sin tener una instancia de ello** . Creé un proyecto en Github para esto, pero aquí está el código:

```
func getTypesOfProperties(in clazz: NSObject.Type) -> Dictionary<String, Any>? {
    var count = UInt32()
    guard let properties = class_copyPropertyList(clazz, &count) else { return nil }
    var types: Dictionary<String, Any> = [:]
    for i in 0..<Int(count) {
        guard let property: objc_property_t = properties[i], let name = getNameOf(property:
property) else { continue }
        let type = getTypeOf(property: property)
        types[name] = type
    }
    free(properties)
    return types
}

func getTypeOf(property: objc_property_t) -> Any {
    guard let attributesAsNSString: NSString = NSString(utf8String:
property_getAttributes(property)) else { return Any.self }
    let attributes = attributesAsNSString as String
    let slices = attributes.components(separatedBy: "\"")
    guard slices.count > 1 else { return getPrimitiveDataType(withAttributes: attributes) }
    let objectClassName = slices[1]
    let objectClass = NSClassFromString(objectClassName) as! NSObject.Type
    return objectClass
}

   func getPrimitiveDataType(withAttributes attributes: String) -> Any {
        guard let letter = attributes.substring(from: 1, to: 2), let type =
primitiveDataTypes[letter] else { return Any.self }
        return type
    }
```

Donde primitiveDataTypes es un Diccionario que asigna una letra en la cadena de atributo a un

tipo de valor:

```
let primitiveDataTypes: Dictionary<String, Any> = [
    "c" : Int8.self,
    "s" : Int16.self,
    "i" : Int32.self,
    "q" : Int.self, //also: Int64, NSInteger, only true on 64 bit platforms
    "S" : UInt16.self,
    "I" : UInt32.self,
    "Q" : UInt.self, //also UInt64, only true on 64 bit platforms
    "B" : Bool.self,
    "d" : Double.self,
    "f" : Float.self,
    "{" : Decimal.self
]

   func getNameOf(property: objc_property_t) -> String? {
        guard let name: NSString = NSString(utf8String: property_getName(property)) else {
return nil }
        return name as String
    }
```

Puede extraer el NSObject.Type de todas las propiedades que tipo de clase hereda de NSObject como NSDate (Swift3: Date), NSString (Swift3: String ?) Y NSNumber , sin embargo, se almacena en el tipo Any (como se puede ver como Tipo del valor del diccionario devuelto por el método). Esto se debe a las limitaciones de los value types de value types , como Int, Int32, Bool. Desde esos tipos no heredan de NSObject, llamando .self sobre, por ejemplo un Int - Int.self no vuelve NSObject.Type, sino más bien del tipo Any . Por lo tanto, el método devuelve Dictionary<String, Any>? y no Dictionary<String, NSObject.Type>? .

Puedes usar este método así:

```
class Book: NSObject {
    let title: String
    let author: String?
    let numberOfPages: Int
    let released: Date
    let isPocket: Bool

    init(title: String, author: String?, numberOfPages: Int, released: Date, isPocket: Bool) {
        self.title = title
        self.author = author
        self.numberOfPages = numberOfPages
        self.released = released
        self.isPocket = isPocket
    }
}

guard let types = getTypesOfProperties(in: Book.self) else { return }
for (name, type) in types {
    print("'\(name)' has type '\(type)'")
}
// Prints:
// 'title' has type 'NSString'
// 'numberOfPages' has type 'Int'
// 'author' has type 'NSString'
// 'released' has type 'NSDate'
// 'isPocket' has type 'Bool'
```

También puede intentar convertir el Any en NSObject.Type , que tendrá éxito para todas las propiedades NSObject de NSObject , luego puede verificar el tipo utilizando el operador estándar == :

```
func checkPropertiesOfBook() {
    guard let types = getTypesOfProperties(in: Book.self) else { return }
    for (name, type) in types {
        if let objectType = type as? NSObject.Type {
            if objectType == NSDate.self {
                print("Property named '\(name)' has type 'NSDate'")
            } else if objectType == NSString.self {
                print("Property named '\(name)' has type 'NSString'")
            }
        }
    }
}
```

Si declara este operador personalizado `==` :

```
func ==(rhs: Any, lhs: Any) -> Bool {
    let rhsType: String = "\(rhs)"
    let lhsType: String = "\(lhs)"
    let same = rhsType == lhsType
    return same
}
```

Incluso puedes verificar el tipo de value types de value types como este:

```
func checkPropertiesOfBook() {
    guard let types = getTypesOfProperties(in: Book.self) else { return }
    for (name, type) in types {
        if type == Int.self {
            print("Property named '\(name)' has type 'Int'")
        } else if type == Bool.self {
            print("Property named '\(name)' has type 'Bool'")
        }
    }
}
```

LIMITACIONES Esta solución no funciona cuando los value types son opcionales. Si ha declarado una propiedad en su subclase de NSObject como esta: `var myOptionalInt: Int?` , el código anterior no encontrará esa propiedad porque el método `class_copyPropertyList` no contiene tipos de valor opcionales.

Lea Reflexión en línea: https://riptutorial.com/es/swift/topic/1201/reflexion

Capítulo 54: RxSwift

Examples

Conceptos básicos de RxSwift

FRP, o Programación reactiva funcional, tiene algunos términos básicos que debe conocer.

Cada dato puede representarse como Observable , que es un flujo de datos asíncrono. El poder de FRP está en representación de eventos síncronos y asíncronos como flujos, Observable s, y proporciona la misma interfaz para trabajar con él.

Por lo general, Observable tiene varios eventos (o ninguno) que contienen la fecha: eventos .Next , y luego puede terminarse con éxito (.Success) o con un error (.Error).

Echemos un vistazo al siguiente diagrama de mármol:

```
--(1)--(2)--(3)|-->
```

En este ejemplo hay un flujo de valores Int . A medida que el tiempo avanza, se .Next tres eventos .Next y, a continuación, la secuencia terminó correctamente.

```
--X->
```

El diagrama anterior muestra un caso en el que no se emitieron datos y el evento .Error finaliza el Observable .

Antes de continuar, hay algunos recursos útiles:

1. RxSwift . Mira ejemplos, lee documentos y comienza.
2. RxSwift Slack room tiene algunos canales para resolver problemas de educación.
3. Juega con RxMarbles para saber qué hace el operador y cuál es el más útil en tu caso.
4. Echa un vistazo a este ejemplo , explora el código por ti mismo.

Creando observables

RxSwift ofrece muchas formas de crear un Observable , echemos un vistazo:

```
import RxSwift

let intObservale = Observable.just(123) // Observable<Int>
let stringObservale = Observable.just("RxSwift") // Observable<String>
let doubleObservale = Observable.just(3.14) // Observable<Double>
```

Así, se crean los observables. Tienen un solo valor y luego terminan con éxito. Sin embargo, nada sucedió después de que fue creado. ¿Por qué?

Hay dos pasos para trabajar con Observable s: **observas** algo para *crear* una secuencia y luego te **suscribes** a la secuencia o lo **vinculas** a algo para *interactuar* con ella.

```
Observable.just(12).subscribe {
    print($0)
}
```

La consola imprimirá:

```
.Next(12)
.Completed()
```

Y si solo me interesa trabajar con datos, que tienen lugar en eventos .Next , usaría el operador subscribeNext :

```
Observable.just(12).subscribeNext {
    print($0) // prints "12" now
}
```

Si quiero crear un observable de muchos valores, uso diferentes operadores:

```
Observable.of(1,2,3,4,5).subscribeNext {
    print($0)
}
// 1
// 2
// 3
// 4
// 5

// I can represent existing data types as Observables also:
[1,2,3,4,5].asObservable().subscribeNext {
    print($0)
}
// result is the same as before.
```

Y finalmente, tal vez quiero un Observable que haga algo de trabajo. Por ejemplo, es conveniente envolver una operación de red en Observable<SomeResultType> . Echemos un vistazo a uno puede lograr esto:

```
Observable.create { observer in    // create an Observable ...
    MyNetworkService.doSomeWorkWithCompletion { (result, error) in
        if let e = error {
            observer.onError(e)    // ..that either holds an error
        } else {
            observer.onNext(result) // ..or emits the data
            observer.onCompleted()  // ..and terminates successfully.
        }
    }
    return NopDisposable.instance // here you can manually free any resources
                                //in case if this observable is being disposed.
}
```

Desechando

Una vez que se creó la suscripción, es importante administrar su desasignación correcta.

Los docs nos dijeron que

> Si una secuencia termina en un tiempo finito, no llamar a dispose o no usar addDisposableTo (disposeBag) no causará ninguna fuga permanente de recursos. Sin embargo, esos recursos se utilizarán hasta que la secuencia se complete, ya sea terminando la producción de elementos o devolviendo un error.

Hay dos formas de desasignar recursos.

1. Usando disposeBag sy el operador addDisposableTo .
2. Utilizando el operador takeUntil .

En el primer caso, usted pasa manualmente la suscripción al objeto DisposeBag , que borra correctamente toda la memoria tomada.

```
let bag = DisposeBag()
Observable.just(1).subscribeNext {
    print($0)
}.addDisposableTo(bag)
```

En realidad, no es necesario crear DisposeBag s en cada clase que cree, solo eche un vistazo al *proyecto de la Comunidad* RxSwift llamado NSObject + Rx . Usando el marco, el código anterior puede reescribirse como sigue:

```
Observable.just(1).subscribeNext {
    print($0)
}.addDisposableTo(rx_disposeBag)
```

En el segundo caso, si el tiempo de suscripción coincide con el self duración del objeto, es posible implementar la eliminación usando takeUntil(rx_deallocated) :

```
let _ = sequence
    .takeUntil(rx_deallocated)
    .subscribe {
        print($0)
    }
```

Fijaciones

```
Observable.combineLatest(firstName.rx_text, lastName.rx_text) { $0 + " " + $1 }
.map { "Greetings, \($0)" }
.bindTo(greetingLabel.rx_text)
```

Usando el operador combineLatest cada vez que un elemento sea emitido por cualquiera de los dos Observables , combine el último elemento emitido por cada Observable . Entonces, de esta manera, combinamos el resultado de los dos UITextField 's creando un nuevo mensaje con el texto "Greetings, \($0)" utilizando la interpolación de cadenas para enlazar más tarde al texto de un UILabel .

Podemos enlazar datos a cualquier UITableView y UICollectionView de una manera muy fácil:

```
viewModel
.rows
.bindTo(resultsTableView.rx_itemsWithCellIdentifier("WikipediaSearchCell", cellType:
WikipediaSearchCell.self)) { (_, viewModel, cell) in
    cell.title = viewModel.title
    cell.url = viewModel.url
}
.addDisposableTo(disposeBag)
```

Eso es un envoltorio Rx alrededor del método de fuente de datos cellForRowAtIndexPath . Y también Rx se encarga de la implementación de numberOfRowsAtIndexPath , que es un método requerido en un sentido tradicional, pero no tiene que implementarlo aquí, se cuida.

RxCocoa y ControlEvents

RxSwift proporciona no solo las formas de controlar sus datos, sino que también representa las acciones de los usuarios de manera reactiva.

RxCocoa contiene todo lo que necesitas. Envuelve la mayoría de las propiedades de los componentes de la interfaz de usuario en Observable s, pero no realmente. Hay algunos Observable actualizados llamados ControlEvent s (que representan eventos) y ControlProperties (que representan propiedades, ¡sorpresa!). Estas cosas contienen corrientes Observable bajo el capó, pero también tienen algunos matices:

- Nunca falla, así que no hay errores.
- Se Complete secuencia de control desasignada.
- Ofrece eventos en el hilo principal (MainScheduler.instance).

Básicamente, puedes trabajar con ellos como de costumbre:

```
button.rx_tap.subscribeNext { _ in   // control event
    print("User tapped the button!")
}.addDisposableTo(bag)

textField.rx_text.subscribeNext { text in // control property
    print("The textfield contains: \(text)")
}.addDisposableTo(bag)
// notice that ControlProperty generates .Next event on subscription
// In this case, the log will display
// "The textfield contains: "
// at the very start of the app.
```

Esto es muy importante de usar: siempre que uses Rx, olvídate de las cosas de @IBAction , todo lo que necesites se puede vincular y configurar a la vez. Por ejemplo, el método viewDidLoad de su controlador de vista es un buen candidato para describir cómo funcionan los componentes de la interfaz de usuario.

Ok, otro ejemplo: supongamos que tenemos un campo de texto, un botón y una etiqueta. Queremos **validar el texto** en el campo de texto cuando nos **toque** el botón y **mostrar** los resultados en la etiqueta. Sí, parece otra tarea de validar correo electrónico, ¿eh?

En primer lugar, tomamos el button.rx_tap ControlEvent:

```
----()-----------------------()----->
```

Aquí los paréntesis vacíos muestran los grifos del usuario. A continuación, tomamos lo que está escrito en el campo de texto con el operador withLatestFrom (withLatestFrom un vistazo aquí , imagine que la secuencia superior representa los toques del usuario, la parte inferior representa el texto en el campo de texto).

```
button.rx_tap.withLatestFrom(textField.rx_text)

----("")--------------------("123")--->
//   ^ tap    ^ i wrote 123     ^ tap
```

Bien, tenemos una secuencia de cadenas para validar, emitidas solo cuando necesitamos validar.

Cualquier Observable tiene operadores tan familiares como map o filter , tomaremos el map para validar el texto. Cree la función validateEmail usted mismo, use cualquier expresión regular que desee.

```
button.rx_tap                                // ControlEvent<Void>
        .withLatestFrom(textField.rx_text)   // Observable<String>
        .map(validateEmail)                  // Observable<Bool>
        .map { (isCorrect) in
            return isCorrect ? "Email is correct" : "Input the correct one, please"
        }                                    // Observable<String>
        .bindTo(label.rx_text)
        .addDisposableTo(bag)
```

¡Hecho! Si necesita más lógica personalizada (como mostrar vistas de error en caso de error, hacer una transición a otra pantalla en caso de éxito ...), simplemente suscríbase a la secuencia final de Bool y escríbala allí.

Lea RxSwift en línea: https://riptutorial.com/es/swift/topic/4890/rxswift

Capítulo 55: Servidor HTTP Swift de Kitura

Introducción

Servidor Swift con Kitura

Kitura es un framework web escrito en swift que se crea para servicios web. Es muy fácil de configurar para solicitudes HTTP. Para el entorno, necesita OS X con XCode instalado o Linux ejecutando swift 3.0.

Examples

Hola aplicación mundial

Configuración

Primero, crea un archivo llamado Package.swift. Este es el archivo que le dice al compilador de Swift dónde están ubicadas las bibliotecas. En este ejemplo de hello world, estamos usando los repositorios de GitHub. Necesitamos Kitura y HeliumLogger . Ponga el siguiente código dentro de Package.swift. Especificó el nombre del proyecto como *kitura-helloworld* y también las direcciones URL de dependencia.

```
import PackageDescription
let package = Package(
    name: "kitura-helloworld",
        dependencies: [
            .Package(url: "https://github.com/IBM-Swift/HeliumLogger.git", majorVersion: 1,
minor: 6),
            .Package(url: "https://github.com/IBM-Swift/Kitura.git", majorVersion: 1, minor:
6) ] )
```

A continuación, cree una carpeta llamada Fuentes. Dentro, crea un archivo llamado main.swift. Este es el archivo que implementamos toda la lógica para esta aplicación. Ingrese el siguiente código en este archivo principal.

Importar bibliotecas y habilitar el registro.

```
import Kitura
import Foundation
import HeliumLogger

HeliumLogger.use()
```

Añadiendo un enrutador. El enrutador especifica una ruta, tipo, etc. de la solicitud HTTP. Aquí estamos agregando un controlador de solicitud GET que imprime *Hello world* , y luego una solicitud de publicación que lee texto sin formato de la solicitud y luego lo devuelve.

```
let router = Router()

router.get("/get") {
    request, response, next in
    response.send("Hello, World!")
    next()
}

router.post("/post") {
    request, response, next in
    var string: String?
    do{
```

```
        string = try request.readString()

    } catch let error {
        string = error.localizedDescription
    }
    response.send("Value \(string!) received.")
    next()
}
```

Especifique un puerto para ejecutar el servicio.

```
let port = 8080
```

Enlazar el enrutador y el puerto y agregarlos como servicio HTTP

```
Kitura.addHTTPServer(onPort: port, with: router)
Kitura.run()
```

Ejecutar

Navegue a la carpeta raíz con el archivo Package.swift y la carpeta Resources. Ejecute el siguiente comando. El compilador Swift descargará automáticamente los recursos mencionados en Package.swift en la carpeta Paquetes y luego compilará estos recursos con main.swift

```
swift build
```

Cuando la compilación haya finalizado, el ejecutable se colocará en esta ubicación. Haga doble clic en este ejecutable para iniciar el servidor.

```
.build/debug/kitura-helloworld
```

Validar

Abra un navegador, escriba localhost:8080/get como url y presione enter. La página de hola mundo debería salir.

Abra una aplicación de solicitud HTTP, publique texto sin formato en localhost:8080/post . La cadena de respuesta mostrará el texto introducido correctamente.

Lea Servidor HTTP Swift de Kitura en línea: https://riptutorial.com/es/swift/topic/10690/servidor-http-swift-de-kitura

Capítulo 56: Tipo de fundición

Sintaxis

- let name = json["name"] as? String ?? "" // Salida: john
- let name = json["name"] as? String // Output: Optional("john")
- let name = rank as? Int // Output: Optional(1)
- let name = rank as? Int ?? 0 // Output: 1
- let name = dictionary as? [String: Any] ?? [:] // Output: ["name" : "john", "subjects": ["Maths", "Science", "English", "C Language"]]

Examples

Downcasting

Una variable puede reducirse a un subtipo utilizando los *operadores de* conversión de *tipos* as? , y as! .

El as? El operador *intenta* lanzar un subtipo.
Puede fallar, por lo tanto devuelve un opcional.

```
let value: Any = "John"

let name = value as? String
print(name) // prints Optional("John")

let age = value as? Double
print(age) // prints nil
```

El as! El operador *fuerza* un yeso.
No devuelve un opcional, pero se bloquea si falla la conversión.

```
let value: Any = "Paul"

let name = value as! String
print(name) // prints "Paul"

let age = value as! Double // crash: "Could not cast value…"
```

Es común usar operadores de conversión de tipos con desenvolvimiento condicional:

```
let value: Any = "George"

if let name = value as? String {
    print(name) // prints "George"
}

if let age = value as? Double {
    print(age) // Not executed
}
```

Casting con interruptor

La instrucción de switch también se puede utilizar para intentar convertir en diferentes tipos:

```
func checkType(_ value: Any) -> String {
    switch value {

    // The `is` operator can be used to check a type
    case is Double:
        return "value is a Double"

    // The `as` operator will cast. You do not need to use `as?` in a `switch`.
    case let string as String:
        return "value is the string: \(string)"

    default:
        return "value is something else"
    }

}

checkType("Cadena")  // "value is the string: Cadena"
checkType(6.28)      // "value is a Double"
checkType(UILabel()) // "value is something else"
```

Upcasting

El operador as se convertirá en un supertipo. Como no puede fallar, no devuelve un opcional.

```
let name = "Ringo"
let value = string as Any  // `value` is of type `Any` now
```

Ejemplo de uso de un downcast en un parámetro de función que involucra subclases

Se puede utilizar un downcast para hacer uso del código y los datos de una subclase dentro de una función que toma un parámetro de su superclase.

```
class Rat {
    var color = "white"
}

class PetRat: Rat {
    var name = "Spot"
}

func nameOfRat(: Rat) -> String {
    guard let petRat = ( as? PetRat) else {
        return "No name"
    }

    return petRat.name
}

let noName = Rat()
let spot = PetRat()

print(nameOfRat(noName))
print(nameOfRat(spot))
```

Tipo casting en Swift Language

Tipo de fundición

La conversión de tipos es una forma de verificar el tipo de una instancia, o de tratar esa instancia como una superclase o subclase diferente de otro lugar en su propia jerarquía de clases.

La conversión de tipos en Swift se implementa con los operadores de y como. Estos dos operadores proporcionan una forma simple y expresiva de verificar el tipo de un valor o convertir un valor en un tipo diferente.

Downcasting

Una constante o variable de un cierto tipo de clase puede en realidad referirse a una instancia de una subclase detrás de escena. Cuando crea que este es el caso, puede intentar reducir al tipo de subclase con un operador de conversión de tipo (como? O como!).

Debido a que el downcasting puede fallar, el operador de conversión de tipo viene en dos formas diferentes. La forma condicional, as ?, devuelve un valor opcional del tipo que está intentando reducir. La forma forzada, as !, intenta abatir y forzar el resultado como una acción compuesta única.

Utilice la forma condicional del operador de conversión de tipo (¿como?) Cuando no esté seguro de si el derrumbe tendrá éxito. Esta forma del operador siempre devolverá un valor opcional, y el valor será nulo si no fuera posible la reducción. Esto le permite verificar un descenso exitoso.

Use la forma forzada del operador de conversión de tipos (como!) Solo cuando esté seguro de que el retroceso siempre tendrá éxito. Esta forma del operador desencadenará un error de tiempo de ejecución si intenta reducir a un tipo de clase incorrecto. Saber más.

Conversión de cadena a Int & Float: -

```
let numbers = "888.00"
let intValue = NSString(string: numbers).integerValue
print(intValue) // Output - 888


let numbers = "888.00"
let floatValue = NSString(string: numbers).floatValue
print(floatValue) // Output : 888.0
```

Conversión de flotar a cadena

```
    let numbers = 888.00
    let floatValue = String(numbers)
    print(floatValue) // Output : 888.0

    // Get Float value at particular decimal point
    let numbers = 888.00
    let floatValue = String(format: "%.2f", numbers) // Here %.2f will give 2 numbers after
decimal points we can use as per our need
    print(floatValue) // Output : "888.00"
```

Entero a valor de cadena

```
let numbers = 888
let intValue = String(numbers)
print(intValue) // Output : "888"
```

Flotante a valor de cadena

```
let numbers = 888.00
let floatValue = String(numbers)
print(floatValue)
```

Valor flotante opcional a la cadena

```
let numbers: Any = 888.00
let floatValue = String(describing: numbers)
print(floatValue) // Output : 888.0
```

Cadena opcional a valor de Int

```
let hitCount = "100"
let data :AnyObject = hitCount
let score = Int(data as? String ?? "") ?? 0
print(score)
```

Disminuyendo los valores de JSON

```
    let json = ["name" : "john", "subjects": ["Maths", "Science", "English", "C Language"]] as
[String : Any]
    let name = json["name"] as? String ?? ""
    print(name) // Output : john
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects) // Output : ["Maths", "Science", "English", "C Language"]
```

Disminución de valores desde JSON opcional

```
    let response: Any = ["name" : "john", "subjects": ["Maths", "Science", "English", "C
Language"]]
    let json = response as? [String: Any] ?? [:]
    let name = json["name"] as? String ?? ""
    print(name) // Output : john
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects) // Output : ["Maths", "Science", "English", "C Language"]
```

Gestionar JSON Response con condiciones.

```
    let response: Any = ["name" : "john", "subjects": ["Maths", "Science", "English", "C
Language"]] //Optional Response

    guard let json = response as? [String: Any] else {
```

```
        // Handle here nil value
        print("Empty Dictionary")
        // Do something here
        return
    }
    let name = json["name"] as? String ?? ""
    print(name) // Output : john
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects) // Output : ["Maths", "Science", "English", "C Language"]
```

Gestionar respuesta nula con condición

```
    let response: Any? = nil
    guard let json = response as? [String: Any] else {
        // Handle here nil value
        print("Empty Dictionary")
        // Do something here
        return
    }
    let name = json["name"] as? String ?? ""
    print(name)
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects)
```

Salida: `Empty Dictionary`

Lea Tipo de fundición en línea: https://riptutorial.com/es/swift/topic/3082/tipo-de-fundicion

Capítulo 57: Tipografías

Examples

Tipografías para cierres con parámetros.

```
typealias SuccessHandler = (NSURLSessionDataTask, AnyObject?) -> Void
```

Este bloque de código crea un alias de tipo llamado SuccessHandler , de la misma manera que var string = "" crea una variable con la string nombre.

Ahora cada vez que use SuccessHandler , por ejemplo:

```
func example(_ handler: SuccessHandler) {}
```

Usted está escribiendo esencialmente:

```
func example(_ handler: (NSURLSessionDataTask, AnyObject?) -> Void) {}
```

Tipografías para cierres vacíos.

```
typealias Handler = () -> Void
typealias Handler = () -> ()
```

Este bloque crea un alias de tipo que funciona como una función Void to Void (no toma parámetros y no devuelve nada).

Aquí hay un ejemplo de uso:

```
var func: Handler?

func = {}
```

tipografías para otros tipos

```
typealias Number = NSNumber
```

También puede usar un alias de tipo para darle a un tipo otro nombre para que sea más fácil de recordar, o hacer que su código sea más elegante.

tipealias para tuplas

```
typealias PersonTuple = (name: String, age: Int, address: String)
```

Y esto puede ser usado como:

```
func getPerson(for name: String) -> PersonTuple {
    //fetch from db, etc
    return ("name", 45, "address")
}
```

Lea Tipografías en línea: https://riptutorial.com/es/swift/topic/7552/tipografias

Capítulo 58: Trabajando con C y Objective-C

Observaciones

Para obtener más información, consulte la documentación de Apple sobre el uso de Swift con Cocoa y Objective-C .

Examples

Usando clases Swift desde el código Objective-C

En el mismo modulo

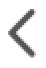
Dentro de un módulo llamado " **MyModule** ", Xcode genera un encabezado llamado **MyModule-Swift.h** que expone las clases públicas de Swift a Objective-C. Importe este encabezado para usar las clases Swift:

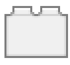
```
// MySwiftClass.swift in MyApp
import Foundation

// The class must be `public` to be visible, unless this target also has a bridging header
public class MySwiftClass: NSObject {
    // ...
}
```

```
// MyViewController.m in MyApp

#import "MyViewController.h"
#import "MyApp-Swift.h"                     // import the generated interface
#import <MyFramework/MyFramework-Swift.h>   // or use angle brackets for a framework target

@implementation MyViewController
- (void)demo {
    [[MySwiftClass alloc] init];  // use the Swift class
}
@end
```

Configuraciones de construcción relevantes:

- **Nombre de encabezado de interfaz generado por Objective-C** : controla el nombre del encabezado Obj-C generado.
- **Instale el encabezado de compatibilidad de Objective-C** : si el encabezado -Swift.h debe ser un encabezado público (para los destinos del marco).





En otro modulo

Utilizando **@import MyFramework;** importa todo el módulo, incluidas las interfaces Obj-C a las clases Swift (si la configuración de compilación antes mencionada está habilitada).

Usando las clases de Objective-C del código Swift

Si MyFramework contiene clases de Objective-C en sus encabezados públicos (y el encabezado general), entonces **import MyFramework** es todo lo necesario para usarlos desde Swift.

Puente de encabezados

Un **encabezado puente** hace que las declaraciones de Objective-C y C adicionales sean visibles para el código Swift. Al agregar archivos de proyecto, Xcode puede ofrecer crear un encabezado de puente automáticamente:

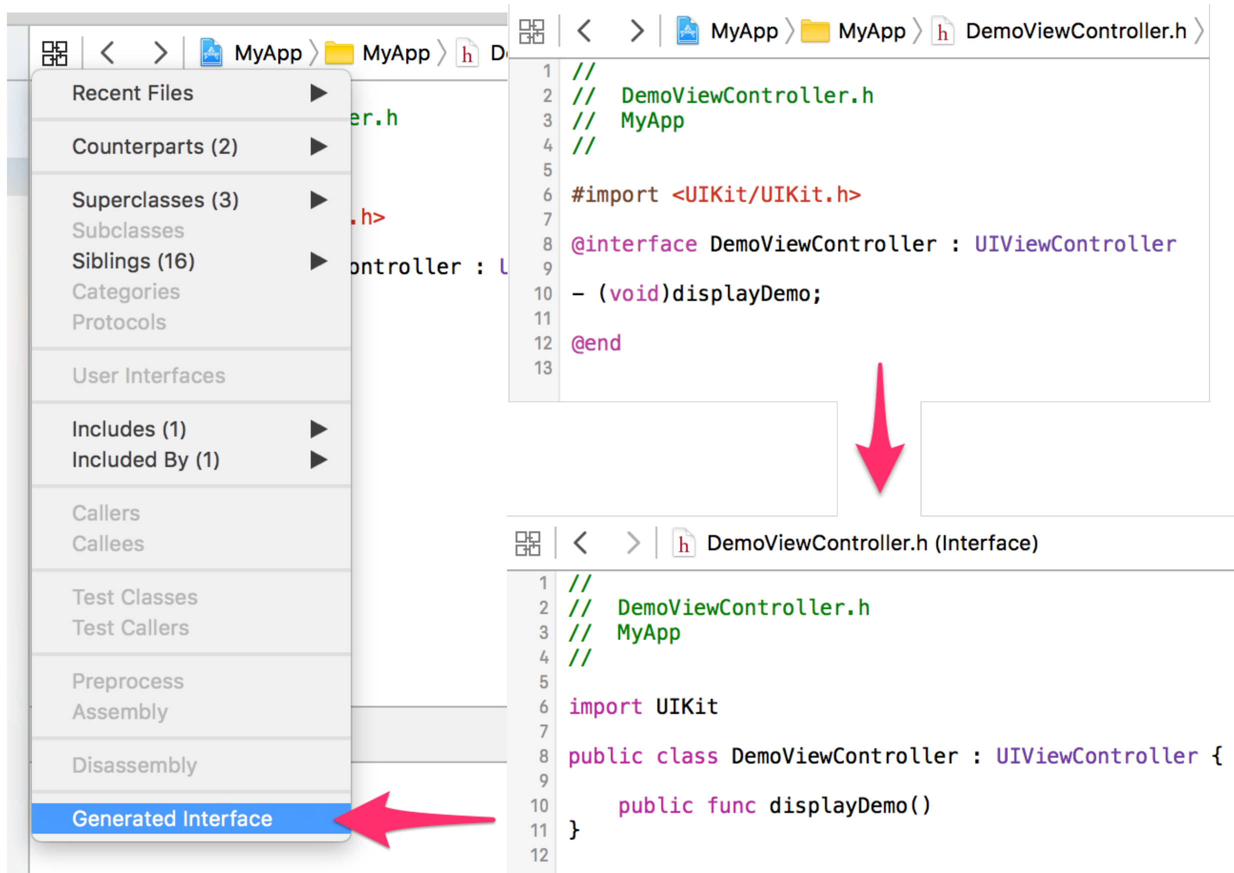
Para crear uno manualmente, modifique la configuración de compilación del **encabezado de puente de Objective-C** :

Dentro del encabezado puente, importe los archivos que sean necesarios para usar desde el código:

```
// MyApp-Bridging-Header.h
#import "MyClass.h"  // allows code in this module to use MyClass
```

Interfaz generada

Haga clic en el botón Elementos relacionados (o presione ^1), luego seleccione **Interfaz generada** para ver la interfaz Swift que se generará desde un encabezado de Objective-C.



Especifique un encabezado de puente para swiftc

El -import-objc-header especifica un encabezado para que swiftc importe:

```
// defs.h
struct Color {
    int red, green, blue;
};

#define MAX_VALUE 255
```

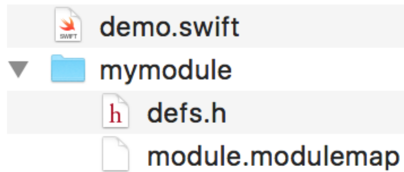
```
// demo.swift
extension Color: CustomStringConvertible {  // extension on a C struct
    public var description: String {
        return "Color(red: \(red), green: \(green), blue: \(blue))"
    }
}
print("MAX_VALUE is: \(MAX_VALUE)")  // C macro becomes a constant
let color = Color(red: 0xCA, green: 0xCA, blue: 0xD0)  // C struct initializer
print("The color is \(color)")
```

```
$ swiftc demo.swift -import-objc-header defs.h && ./demo
MAX_VALUE is: 255
The color is Color(red: 202, green: 202, blue: 208)
```

Usa un mapa de módulo para importar encabezados C

Un **mapa de módulo** puede simplemente import mymodule configurándolo para leer los archivos de encabezado C y hacer que aparezcan como funciones Swift.

Coloque un archivo llamado module.modulemap dentro de un directorio llamado mymodule :



Dentro del archivo del mapa del módulo:

```
// mymodule/module.modulemap
module mymodule {
    header "defs.h"
}
```

Luego import el módulo:

```
// demo.swift
import mymodule
print("Empty color: \(Color())")
```

Utilice el indicador de -I *directory* para indicar a swiftc dónde encontrar el módulo:

```
swiftc -I . demo.swift   # "-I ." means "search for modules in the current directory"
```

Para obtener más información sobre la sintaxis del mapa de módulos, consulte la documentación de Clang sobre mapas de módulos .

Interoperación de grano fino entre Objective-C y Swift

Cuando una API está marcada con NS_REFINED_FOR_SWIFT , tendrá un prefijo con dos guiones bajos (__) cuando se importe a Swift:

```
@interface MyClass : NSObject
- (NSInteger)indexOfObject:(id)obj NS_REFINED_FOR_SWIFT;
@end
```

La interfaz generada se ve así:

```
public class MyClass : NSObject {
    public func __indexOfObject(obj: AnyObject) -> Int
}
```

Ahora puedes **reemplazar la API** con una extensión más "Swifty". En este caso, podemos usar un valor de retorno opcional , filtrando NSNotFound :

```
extension MyClass {
    // Rather than returning NSNotFound if the object doesn't exist,
    // this "refined" API returns nil.
    func indexOfObject(obj: AnyObject) -> Int? {
        let idx = __indexOfObject(obj)
        if idx == NSNotFound { return nil }
        return idx
    }
}

// Swift code, using "if let" as it should be:
let myobj = MyClass()
```

```
if let idx = myobj.indexOfObject(something) {
    // do something with idx
}
```

En la mayoría de los casos, es posible que desee restringir si un argumento a una función de Objective-C podría ser nil . Esto se hace usando la palabra clave _Nonnull , que califica cualquier puntero o referencia de bloque:

```
void
doStuff(const void *const _Nonnull data, void (^_Nonnull completion)())
{
    // complex asynchronous code
}
```

Con eso escrito, el compilador emitirá un error cada vez que intentemos pasar nil a esa función desde nuestro código Swift:

```
doStuff(
    nil,  // error: nil is not compatible with expected argument type 'UnsafeRawPointer'
    nil)  // error: nil is not compatible with expected argument type '() -> Void'
```

El opuesto de _Nonnull es _Nullable , lo que significa que es aceptable pasar nil en este argumento. _Nullable también es el valor predeterminado; sin embargo, especificarlo explícitamente permite un código más auto documentado y preparado para el futuro.

Para ayudar aún más al compilador a optimizar su código, es posible que también desee especificar si el bloque está escapando:

```
void
callNow(__attribute__((noescape)) void (^_Nonnull f)())
{
    // f is not stored anywhere
}
```

Con este atributo, prometemos no guardar la referencia de bloque y no llamar al bloque después de que la función haya finalizado su ejecución.

Usa la biblioteca estándar de C

La interoperabilidad C de Swift le permite usar funciones y tipos de la biblioteca estándar C.

En Linux, la biblioteca estándar de C se expone a través del módulo Glibc ; en las plataformas de Apple se llama Darwin .

```
#if os(macOS) || os(iOS) || os(tvOS) || os(watchOS)
import Darwin
#elseif os(Linux)
import Glibc
#endif

// use open(), read(), and other libc features
```

Lea Trabajando con C y Objective-C en línea:
https://riptutorial.com/es/swift/topic/421/trabajando-con-c-y-objective-c

Capítulo 59: Tuplas

Introducción

Un tipo de tupla es una lista de tipos separados por comas, entre paréntesis.

Esta lista de tipos también puede tener el nombre de los elementos y usar esos nombres para referirse a los valores de los elementos individuales.

Un nombre de elemento consiste en un identificador seguido inmediatamente por dos puntos (:).

Uso común -

Podemos usar un tipo de tupla como el tipo de retorno de una función para permitir que la función devuelva una tupla única que contenga múltiples valores

Observaciones

Las tuplas se consideran tipos de valor. Se puede encontrar más información sobre las tuplas en la documentación: developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.

Examples

¿Qué son las tuplas?

Las tuplas agrupan múltiples valores en un solo valor compuesto. Los valores dentro de una tupla pueden ser de cualquier tipo y no tienen que ser del mismo tipo entre sí.

Las tuplas se crean agrupando cualquier cantidad de valores:

```
let tuple = ("one", 2, "three")

// Values are read using index numbers starting at zero
print(tuple.0) // one
print(tuple.1) // 2
print(tuple.2) // three
```

También se pueden nombrar valores individuales cuando se define la tupla:

```
let namedTuple = (first: 1, middle: "dos", last: 3)

// Values can be read with the named property
print(namedTuple.first)  // 1
print(namedTuple.middle) // dos

// And still with the index number
print(namedTuple.2)      // 3
```

También pueden ser nombrados cuando se usan como una variable e incluso tienen la capacidad de tener valores opcionales dentro de:

```
var numbers: (optionalFirst: Int?, middle: String, last: Int)?

//Later On
numbers = (nil, "dos", 3)

print(numbers.optionalFirst)// nil
```

```
print(numbers.middle)//"dos"
print(numbers.last)//3
```

La descomposición en variables individuales.

Las tuplas se pueden descomponer en variables individuales con la siguiente sintaxis:

```
let myTuple = (name: "Some Name", age: 26)
let (first, second) = myTuple

print(first) // "Some Name"
print(second)  // 26
```

Esta sintaxis se puede usar independientemente de si la tupla tiene propiedades sin nombre:

```
let unnamedTuple = ("uno", "dos")
let (one, two) = unnamedTuple
print(one) // "uno"
print(two) // "dos"
```

Las propiedades específicas se pueden ignorar usando el guión bajo (_):

```
let longTuple = ("ichi", "ni", "san")
let (_, _, third) = longTuple
print(third) // "san"
```

Tuplas como el valor de retorno de las funciones

Las funciones pueden devolver tuplas:

```
func tupleReturner() -> (Int, String) {
    return (3, "Hello")
}

let myTuple = tupleReturner()
print(myTuple.0) // 3
print(myTuple.1) // "Hello"
```

Si asigna nombres de parámetros, pueden usarse desde el valor de retorno:

```
func tupleReturner() -> (anInteger: Int, aString: String) {
    return (3, "Hello")
}

let myTuple = tupleReturner()
print(myTuple.anInteger) // 3
print(myTuple.aString) // "Hello"
```

Usando una tipografía para nombrar tu tipo de tupla

En ocasiones, es posible que desee utilizar el mismo tipo de tupla en varios lugares a lo largo de su código. Esto puede ensuciarse rápidamente, especialmente si tu tupla es compleja:

```
// Define a circle tuple by its center point and radius
let unitCircle: (center: (x: CGFloat, y: CGFloat), radius: CGFloat) = ((0.0, 0.0), 1.0)
```

```
func doubleRadius(ofCircle circle: (center: (x: CGFloat, y: CGFloat), radius: CGFloat)) ->
(center: (x: CGFloat, y: CGFloat), radius: CGFloat) {
    return (circle.center, circle.radius * 2.0)
}
```

Si usa un determinado tipo de tupla en más de un lugar, puede usar la palabra clave typealias para nombrar su tipo de tupla.

```
// Define a circle tuple by its center point and radius
typealias Circle = (center: (x: CGFloat, y: CGFloat), radius: CGFloat)

let unitCircle: Circle = ((0.0, 0.0), 1)

func doubleRadius(ofCircle circle: Circle) -> Circle {
    // Aliased tuples also have access to value labels in the original tuple type.
    return (circle.center, circle.radius * 2.0)
}
```

Sin embargo, si te encuentras haciendo esto con demasiada frecuencia, debes considerar usar una struct lugar.

Intercambiando valores

Las tuplas son útiles para intercambiar valores entre 2 (o más) variables sin usar variables temporales.

Ejemplo con 2 variables

Dadas 2 variables

```
var a = "Marty McFly"
var b = "Emmett Brown"
```

Podemos intercambiar fácilmente los valores.

```
(a, b) = (b, a)
```

Resultado:

```
print(a) // "Emmett Brown"
print(b) // "Marty McFly"
```

Ejemplo con 4 variables

```
var a = 0
var b = 1
var c = 2
var d = 3

(a, b, c, d) = (d, c, b, a)

print(a, b, c, d) // 3, 2, 1, 0
```

Tuplas como caso en el interruptor

Usa tuplas en un interruptor

```
let switchTuple = (firstCase: true, secondCase: false)

switch switchTuple {
 case (true, false):
    // do something
 case (true, true):
    // do something
 case (false, true):
    // do something
 case (false, false):
    // do something
}
```

O en combinación con un Enum Por ejemplo con clases de tamaño:

```
    let switchTuple = (UIUserInterfaceSizeClass.Compact, UIUserInterfaceSizeClass.Regular)

    switch switchTuple {
    case (.Regular, .Compact):
        //statement
    case (.Regular, .Regular):
        //statement
    case (.Compact, .Regular):
        //statement
    case (.Compact, .Compact):
        //statement
    }
```

Lea Tuplas en línea: https://riptutorial.com/es/swift/topic/574/tuplas

Capítulo 60: Variables y propiedades

Observaciones

Propiedades : Asociado a un tipo.

Variables : No asociadas a un tipo.

Consulte el iBook del lenguaje de programación Swift para obtener más información.

Examples

Creando una Variable

Declare una nueva variable con var , seguido de un nombre, tipo y valor:

```
var num: Int = 10
```

Las variables pueden tener sus valores cambiados:

```
num = 20 // num now equals 20
```

A menos que se definan con let :

```
let num: Int = 10 // num cannot change
```

Swift infiere el tipo de variable, por lo que no siempre tiene que declarar el tipo de variable:

```
let ten = 10 // num is an Int
let pi = 3.14 // pi is a Double
let floatPi: Float = 3.14 // floatPi is a Float
```

Los nombres de variables no están restringidos a letras y números, también pueden contener la mayoría de los otros caracteres Unicode, aunque hay algunas restricciones

> Los nombres de constante y variable no pueden contener caracteres de espacio en blanco, símbolos matemáticos, flechas, puntos de código Unicode de uso privado (o no válido) o caracteres de dibujo de líneas y cuadros. Tampoco pueden comenzar con un número.
>
> **Fuente** developer.apple.com

```
var π: Double = 3.14159
var □□: String = "Apples"
```

Conceptos básicos de propiedad

Las propiedades se pueden agregar a una clase o estructura (técnicamente también enumerados , vea el ejemplo "Propiedades calculadas "). Estos agregan valores que se asocian con instancias de clases / estructuras:

```
class Dog {
    var name = ""
}
```

En el caso anterior, las instancias de Dog tienen una propiedad llamada name de tipo String . Se puede acceder y modificar la propiedad en instancias de Dog :

```
let myDog = Dog()
myDog.name = "Doggy" // myDog's name is now "Doggy"
```

Estos tipos de propiedades se consideran **propiedades almacenadas** , ya que almacenan algo en un objeto y afectan su memoria.

Propiedades almacenadas perezosas

Las propiedades almacenadas perezosas tienen valores que no se calculan hasta que se accede por primera vez. Esto es útil para guardar memoria cuando el cálculo de la variable es computacionalmente costoso. Usted declara una propiedad lazy con lazy :

```
lazy var veryExpensiveVariable = expensiveMethod()
```

A menudo se asigna a un valor de retorno de un cierre:

```
lazy var veryExpensiveString = { () -> String in
    var str = expensiveStrFetch()
    str.expensiveManipulation(integer: arc4random_uniform(5))
    return str
}()
```

Las propiedades almacenadas perezosas deben ser declaradas con var .

Propiedades calculadas

A diferencia de las propiedades almacenadas, las propiedades **computadas** se construyen con un getter y un setter, realizando el código necesario cuando se accede y se establece. Las propiedades calculadas deben definir un tipo:

```
var pi = 3.14

class Circle {
    var radius = 0.0
    var circumference: Double {
        get {
            return pi * radius * 2
        }
        set {
            radius = newValue / pi / 2
        }
    }
}

let circle = Circle()
circle.radius = 1
print(circle.circumference) // Prints "6.28"
circle.circumference = 14
print(circle.radius) // Prints "2.229..."
```

Una propiedad calculada de solo lectura todavía se declara con una var :

```
var circumference: Double {
    get {
        return pi * radius * 2
    }
}
```

Las propiedades calculadas de solo lectura se pueden acortar para excluir get :

```
var circumference: Double {
    return pi * radius * 2
}
```

Variables locales y globales

Las variables locales se definen dentro de una función, método o cierre:

```
func printSomething() {
    let localString = "I'm local!"
    print(localString)
}

func printSomethingAgain() {
    print(localString) // error
}
```

Las variables globales se definen fuera de una función, método o cierre, y no se definen dentro de un tipo (piense fuera de todos los corchetes). Se pueden utilizar en cualquier lugar:

```
let globalString = "I'm global!"
print(globalString)

func useGlobalString() {
    print(globalString) // works!
}

for i in 0..<2 {
    print(globalString) // works!
}

class GlobalStringUser {
    var computeGlobalString {
        return globalString // works!
    }
}
```

Las variables globales se definen perezosamente (vea el ejemplo de "Propiedades perezosas").

Tipo de propiedades

Las propiedades de tipo son propiedades en el tipo en sí, no en la instancia. Pueden ser tanto propiedades almacenadas como computadas. Usted declara una propiedad de tipo con static :

```
struct Dog {
    static var noise = "Bark!"
}

print(Dog.noise) // Prints "Bark!"
```

En una clase, puede usar la palabra clave de class lugar de static para hacerla reemplazable. Sin embargo, solo puede aplicar esto en las propiedades computadas:

```
class Animal {
    class var noise: String {
        return "Animal noise!"
    }
}
```

```
class Pig: Animal {
    override class var noise: String {
        return "Oink oink!"
    }
}
```

Esto se utiliza a menudo con el patrón de singleton .

Observadores de la propiedad

Los observadores de propiedades responden a cambios en el valor de una propiedad.

```
var myProperty = 5 {
    willSet {
        print("Will set to \(newValue). It was previously \(myProperty)")
    }
    didSet {
        print("Did set to \(myProperty). It was previously \(oldValue)")
    }
}
myProperty = 6
// prints: Will set to 6, It was previously 5
// prints: Did set to 6. It was previously 5
```

- willSet se llama **antes de** myProperty se establece. El nuevo valor está disponible como newValue , y el valor anterior todavía está disponible como myProperty .
- didSet se llama **después de establecer** myProperty . El valor de edad está disponible como oldValue , y el nuevo valor ya está disponible como myProperty .

  Nota: didSet y willSet no se willSet en los siguientes casos:

  - Asignando un valor inicial
  - Modificando la variable dentro de su propio didSet o willSet

- Los nombres de parámetros para oldValue y newValue de didSet y willSet también se pueden declarar para aumentar la legibilidad:

```
var myFontSize = 10 {
    willSet(newFontSize) {
        print("Will set font to \(newFontSize), it was \(myFontSize)")
    }
    didSet(oldFontSize) {
        print("Did set font to \(myFontSize), it was \(oldFontSize)")
    }
}
```

  Precaución: si bien se admite la declaración de nombres de parámetros de establecimiento, se debe tener cuidado de no mezclar los nombres:

  - willSet(oldValue) y didSet(newValue) son completamente legales, pero confundirán considerablemente a los lectores de su código.

Lea Variables y propiedades en línea: https://riptutorial.com/es/swift/topic/536/variables-y-propiedades

Creditos

S. No	Capítulos	Contributors
1	Empezando con Swift Language	Ahmad F, Anas, andy, Cailean Wilkinson, Claw, Community, esthepiking, Ferenc Kiss, Jim, jtbandes, Luca Angeletti, Luca Angioloni, Moritz, nmnsud, Seyyed Parsa Neshaei, sudo, Sunil Prajapati, Tanner, user3581248
2	(Inseguro) punteros de búfer	Tommie C.
3	Actuación	Matthew Seaman
4	Algoritmos con Swift	Austin Conlon, Bohdan Savych, Hady Nourallah, SteBra, Stephen Leppik, Tommie C.
5	Almacenamiento en caché en el espacio en disco	Viktor Gardart
6	Arrays	BaSha, Ben Trengrove, D4ttatraya, DarkDust, Hamish, jtbandes, Kevin, Luca Angeletti, Moritz, Moriya, nathan, pableiros, Palle , Saagar Jha, Stephen Leppik, ThrowingSpoon, tomahh, toofani, vacawama, Vladimir Nul
7	Bloques	Matt
8	Booleanos	Andreas, jtbandes, Kevin, pableiros
9	Bucles	Caleb Kleveter, D31, Efraim Weiss, Fred Faust, Hamish, Idan, Irfan, Jeff Lewis, Luca Angeletti, Moritz, Mr. Xcoder, Saagar Jha, Santa Claus, WMios, xoudini
10	Cambiar	Ajwhiteway, AK1, Duncan C, elprl, Harshal Bhavsar, joan, Josh Brown, Luca Angeletti, Moritz, Santa Claus, ThrowingSpoon
11	Cierres	ctietze, Duncan C, Hamish, Jojodmo, jtbandes, LopSae, Matthew Seaman, Moritz, Timothy Rascher, Tom Magnusson
12	Cifrado AES	Matt, Stephen Leppik, zaph
13	Comenzando con la Programación Orientada al Protocolo	Alessandro Orrù, Fred Faust, kabiroberai, Krzysztof Romanowski
14	Concurrencia	Adda_25, Ahmad F, FelixSFD, JAL, LukeSideWalker, M_G, Matthew Seaman, Palle, Rob, Santa Claus
15	Condicionales	AK1, atxe, Brduca, Community, Dalija Prasnikar, DarkDust, Hamish, jtbandes, ThaNerd, Thomas Gerot, tktsubota, toofani, torinpitchers
16	Conjuntos	Community, Dalija Prasnikar, Luca Angeletti, Moritz, Steve Moser
17	Control de acceso	4444, Asdrubal, FelixSFD

18	Controlador de finalización	Maysam, Moritz
19	Convenciones de estilo	Grimxn, Moritz, Palle, Ryan H.
20	Cuerdas y personajes	Akshit Soota, Andrea Antonioni, antonio081014, AstroCB, Caleb Kleveter, Carpsen90, egor.zhdan, Feldur, Franck Dernoncourt, Govind Rai, Greg, Guilherme Torres Castro, Hamish, HariKrishnan.P, HeMet, JAL, Jason Sturges, Jojodmo, jtbandes, kabiroberai, Kirit Modi, Kyle KIM, Lope, LopSae, Luca Angeletti , LukeSideWalker, Magisch, Mahmoud Adam, Matt, Matthew Seaman, Max Desiatov, maxkonovalov, Moritz, Nate Cook, Nikolai Ruhe, Panda, Patrick, pixatlazaki, QoP, sdasdadas, Shanmugaraja G, shim, solidcell, Sunil Sharma, Suragch, taylor swift, The_Curry_Man, ThrowingSpoon, user3480295, Victor Sigler, Vinupriya Arivazhagan, WMios
21	Derivación de clave PBKDF2	BUZZE, zaph
22	Entrar en Swift	Adam Bardon, D4ttatraya, DanHabib, jglasse, Moritz, paper1111, RamenChef
23	Enums	Alex Popov, Anh Pham, Avi, Caleb Kleveter, Diogo Antunes, Fantattitude, fredpi, Hamish, Jason Sturges, Jojodmo, jtbandes, juanjo, Justin Whitney, Matt, Matthew Seaman, Nathan Kellert, Nick Podratz, Nikolai Ruhe, SeanRobinson159, shannoga, user3480295
24	Estructuras	Accepted Answer, AK1, Diogo Antunes, fredpi, Josh Brown, Kevin, Luca Angeletti, Marcus Rossel, Moritz, pbush25, Rob Napier, SamG
25	Extensiones	Brduca, David, Esqarrouth, Jojodmo, jtbandes, Luca Angeletti, Moritz, rigdonmr
26	Funcionar como ciudadanos de primera clase en Swift	Kumar Vivek Mitra
27	Funciones	Ajith R Nayak, Andy Ibanez, Caleb Kleveter, jtbandes, Kote, Luca Angeletti, Matt Le Fleur, Nikita Kurtin, noor, ntoonio, Saagar Jha, SKOOP, Stephen Schaub, ThrowingSpoon, tktsubota, ZGski
28	Funciones Swift Advance	DarkDust, Sagar Thummar
29	Generar UIImage de Iniciales desde String	RubberDucky4444
30	Genéricos	Andrey Gordeev, DarkDust, FelixSFD, Glenn R. Fisher, Hamish, Jojodmo, Kent Liau, Luca D'Alberti, Suneet Tipirneni, Ven, xoudini
31	Gestión de la memoria	Accepted Answer, Daniel Firsht, jtbandes, Marc Gravell, Moritz, Palle, Tricertops
32	Gestor de paquetes Swift	Moritz

33	Hash criptográfico	zaph
34	Inicializadores	Brduca, FelixSFD, rashfmnb, Santa Claus, Vinupriya Arivazhagan
35	Inyección de dependencia	Bear with me, JPetric
36	La declaración diferida	Palle
37	Las clases	Dalija Prasnikar, esthepiking, FelixSFD, jtbandes, Luca Angeletti, Matt, Ryan H., tktsubota, Tommie C., Zack
38	Lectura y escritura JSON	Cyril Ivar Garcia, Ethan Kay, Glenn R. Fisher, Ian Rahman, infl3x, Jack C, Jason Sturges, jtbandes, Leo Dabus, lostAtSeaJoshua, Luca D'Alberti, maxkonovalov, Moritz, nstefan, Steffen D. Sommer, Stephen Leppik, toofani
39	Los diccionarios	dasdom, Diogo Antunes, egor.zhdan, iOSDevCenter, Jason Bourne, Kirit Modi, Koushik, Magisch, Moritz, RamenChef, Saagar Jha, sasquatch, Suneet Tipirneni, That lazy iOS Guy ▯, ThrowingSpoon
40	Manejo de errores	Anil Varghese, cpimhoff, egor.zhdan, Jason Bourne, jtbandes, Mehul Sojitra, Moritz, Tom Magnusson
41	Marca de documentación	Abdul Yasin, Martin Delille, Moritz, Rashwan L
42	Método Swizzling	JAL, Noam, Umberto Raimondi
43	NSRegularExpression en Swift	Echelon, Hady Nourallah, ThrowingSpoon
44	Números	Arsen, jtbandes, Suragch, WMios, ZGski
45	Objetos asociados	Fattie, JAL
46	Opcionales	Anand Nimje, Andrey Gordeev, Arnaud, Caleb Kleveter, Hamish, Ian Rahman, iwillnot, Jason Sturges, Jojodmo, juanjo, Kevin, Michaël Azevedo, Moritz, Nathan Kellert, Paulw11, shannoga, SKOOP, Tanner, tktsubota, Tommie C.
47	Operadores Avanzados	avismara, egor.zhdan, Fluidity, Hamish, Intentss, JAL, jtbandes , kennytm, Matthew Seaman, orccrusher99, tharkay
48	OptionSet	4444, Alessandro
49	Patrones de diseño - Creacionales	Ahmad F, AMAN77, Brduca, Dalija Prasnikar, Ian Rahman, Moritz, SeanRobinson159, SimpleBeat, Sơn Đỗ Đình Thy, Stephen Leppik, Thorax, Tommie C.
50	Patrones de diseño - Estructurales	Ian Rahman
51	Programación Funcional en Swift	Echelon, Luca Angeletti, Luke, Matthew Seaman, Shijing Lv
52	Protocolos	Accepted Answer, Ash Furrow, Cory Wilhite, Dalija Prasnikar, esthepiking, Hamish, iBelieve, Igor Bidiniuc, Jason Sturges, Jojodmo, jtbandes, Luca D'Alberti, Matt, matt.baranowski, Matthew Seaman, Oleg Danu, Rahul, SeanRobinson159, SKOOP, Tim

		Vermeulen, tktsubota, Undo, Victor Sigler
53	Reflexión	Asdrubal, LopSae, Sajjon
54	RxSwift	Alexander Olferuk, FelixSFD, imagngames, Moritz, Victor Sigler
55	Servidor HTTP Swift de Kitura	Fangming Ning
56	Tipo de fundición	Anand Nimje, andyvn22, godisgood4, LopSae, Nick Podratz
57	Tipografías	Bartłomiej Semańczyk, Caleb Kleveter, D4ttatraya, Moritz
58	Trabajando con C y Objective-C	4444, Accepted Answer, jtbandes, Mark
59	Tuplas	Accepted Answer, BaSha, Caleb Kleveter, JAL, Jason Sturges, Jojodmo, kabiroberai, LopSae, Luca Angeletti, Moritz, Nathan Kellert, Rick Pasveer, Ronald Martin, tktsubota
60	Variables y propiedades	Christopher Oezbek, FelixSFD, Jojodmo, Luke, Santa Claus, tktsubota