APPRENEZ Swift Language

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#swift

Table des matières

À propos 1
Chapitre 1: Démarrer avec Swift Language 2
Remarques 2
Autres ressources 2
Versions 2
Examples 3
Votre premier programme Swift 3
Installer Swift 4
Votre premier programme dans Swift sur un Mac (en utilisant un terrain de jeu) 5
Votre premier programme dans l'application Swift Playgrounds sur iPad 9
Valeur optionnelle et énumération facultative 12
Chapitre 2: (Unsafe) Buffer Pointers 14
Introduction 14
Remarques 14
Examples 15
UnsafeMutablePointer 15
Cas pratique d'utilisation des tampons 16
Chapitre 3: Algorithmes avec Swift 17
Introduction 17
Examples 17
Tri par insertion 17
Tri 18
Tri de sélection 21
Analyse asymptotique 21
Tri rapide - O (n log n) temps de complexité 22
Graph, Trie, Stack 23
Graphique 23
Trie 31
Empiler 33
Chapitre 4: Balisage de la documentation 37

Examples 37

Documentation de classe 37

Styles de documentation 37

Chapitre 5: Blocs 42

Introduction 42

Examples 42

Fermeture sans fuite 42

Fermeture de fermeture 42

Chapitre 6: Booléens 44

Examples 44

Qu'est-ce que Bool? 44

Annuler un Bool avec le préfixe! opérateur 44

Opérateurs logiques booléens 44

Booléens et conditionnels en ligne 45

Chapitre 7: Boucles 47

Syntaxe 47

Examples 47

Boucle d'introduction 47

Itérer sur une gamme 47

Itérer sur un tableau ou un ensemble 47

Itérer sur un dictionnaire 48

Itérer en sens inverse 48

Itérer sur des plages avec une foulée personnalisée 49

Boucle répétée 50

en boucle 50

Type de séquence pour chaque bloc 50

Boucle d'introduction avec filtrage 51

Briser une boucle 52

Chapitre 8: Casting de type 53

Syntaxe 53

Examples 53

Downcasting 53

Coulée avec interrupteur 54
Upcasting 54
Exemple d'utilisation d'un downcast sur un paramètre de fonction impliquant un sous-classe 54
Type casting en langue rapide 55
Casting de type 55
Downcasting 55
String to Int & Float conversion: - 55
Float to String Conversion 56
Entier à valeur de chaîne 56
Valeur flottante en chaîne 56
Valeur flottante facultative pour la chaîne 56
Chaîne facultative à la valeur Int 56
Valeurs de downcasting de JSON 57
Valeurs de downcasting de JSON facultatif 57
Gérer la réponse JSON avec des conditions 57
Gérer la réponse Nil avec condition 57
Sortie: Empty Dictionary 58
Chapitre 9: Chaînes et caractères 59
Syntaxe 59
Remarques 59
Examples 59
Littérature et caractères 59
Interpolation de chaîne 60
Caractères spéciaux 60
Concaténer des chaînes 61
Examinez et comparez les chaînes 62
Codage et décomposition de chaînes 62
Cordes de décomposition 63
Longueur de la chaîne et itération 63
Unicode 63

Réglage des valeurs 63
Conversions 64
Cordes d'inversion 64
Chaînes majuscules et minuscules 65
Vérifier si la chaîne contient des caractères d'un ensemble défini 65
Compter les occurrences d'un personnage dans une chaîne 67
Supprimer des caractères d'une chaîne non définie dans Set 67
Chaînes de formatage 67
Zéros de tête 67
Nombres après décimal 67
Décimal à hexadécimal 68
Décimal à un nombre avec une base arbitraire 68
Conversion d'une chaîne Swift en un type numérique 68
Itération de chaîne 68
Supprimer les WhiteSpace et NewLine de début et de fin 70
Convertir une chaîne depuis et vers Data / NSData 71
Fractionnement d'une chaîne en un tableau 72
Chapitre 10: Commutateur 73
Paramètres 73
Remarques 73
Examples 73
Utilisation de base 73
Faire correspondre plusieurs valeurs 74
Faire correspondre une plage 74
Utilisation de l'instruction where dans un commutateur 74
Satisfaire l'une des contraintes multiples en utilisant switch 75
Correspondance partielle 75
Changement de vitesse 76
Switch et Enums 77
Commutateur et options 77
Commutateurs et tuples 77
Correspondance basée sur la classe - idéal pour prepareForSegue 78

Chapitre 11: Concurrence 80

Syntaxe 80

Examples 80

Obtention d'une file d'attente Grand Central Dispatch (GCD) 80

Exécution de tâches dans une file d'attente Grand Central Dispatch (GCD) 82

Boucles simultanées 83

Exécution de tâches dans une OperationQueue 84

Création d'opérations de haut niveau 86

Chapitre 12: Conditionnels 89

Introduction 89

Remarques 89

Examples 89

Utiliser la garde 89

Conditions élémentaires: instructions if 90

L'opérateur logique 90

L'opérateur logique OU 90

L'opérateur logique NOT 91

Liaison facultative et clauses "where" 91

Opérateur ternaire 92

Nil-Coalescing Operator 93

Chapitre 13: Contrôle d'accès 94

Syntaxe 94

Remarques 94

Examples 94

Exemple de base utilisant une structure 94

Car.make (public) 95

Car.model (interne) 95

Car.otherName (fichierprivate) 95

Car.fullName (privé) 95

Exemple de sous-classement 95

Exemple de Getters et Setters 96

Chapitre 14: Conventions de style 97

Remarques 97
Examples 97
Effacer l'utilisation 97
Éviter l'ambiguïté 97
Éviter la redondance 97
Nommer les variables en fonction de leur rôle 97
Couplage élevé entre le nom du protocole et les noms de variables 98
Fournir des détails supplémentaires lors de l'utilisation de paramètres faiblement typés 98
Usage Courant 98
Utiliser le langage naturel 98
Méthodes de nommage 98
Paramètres de dénomination dans les initialiseurs et les méthodes d'usine 98
Nommer selon les effets secondaires 99
Fonctions booléennes ou variables 99
Protocoles de nommage 99
Types et propriétés 99
Capitalisation 100
Types et protocoles 100
Tout le reste... 100
Affaire de chameau: 100
Abréviations 100
Chapitre 15: Cryptage AES 102
Examples 102
Cryptage AES en mode CBC avec un IV aléatoire (Swift 3.0) 102
Cryptage AES en mode CBC avec un IV aléatoire (Swift 2.3) 104
Cryptage AES en mode ECB avec remplissage PKCS7 106
Chapitre 16: Dérivation de clé PBKDF2 108
Examples 108
Dérivation de clés par mot de passe 2 (Swift 3) 108
Dérivation de clés par mot de passe 2 (Swift 2.3) 109

Étalonnage par dérivation de clé par mot de passe (Swift 2.3) 110
Calibrage par dérivation de clé par mot de passe (Swift 3) 110
Chapitre 17: Des classes 112
Remarques 112
Examples 112
Définir une classe 112
Sémantique de référence 112
Propriétés et méthodes 113
Classes et héritage multiple 114
deinit 114
Chapitre 18: Design Patterns - Créatif 115
Introduction 115
Examples 115
Singleton 115
Méthode d'usine 115
Observateur 116
Chaîne de responsabilité 117
Itérateur 119
Motif de constructeur 119
Exemple: 120
Continuer: 122
Chapitre 19: Design Patterns - Structural 127
Introduction 127
Examples 127
Adaptateur 127
Façade 127
Chapitre 20: Dictionnaires 129
Remarques 129
Examples 129
Déclaration de dictionnaires 129
Modification des dictionnaires 129
Accès aux valeurs 130

Modifier la valeur du dictionnaire à l'aide de la clé....130
Obtenir toutes les clés dans le dictionnaire....131
Fusionner deux dictionnaires....131
Chapitre 21: Enregistrement dans Swift....132
Remarques....132
Examples....132
Debug Print....132
Mise à jour des valeurs de débogage et d'impression des classes....132
déverser....133
print () vs dump ()....134
imprimer vs NSLog....135
Chapitre 22: Ensembles....136
Examples....136
Déclarations....136
Modification de valeurs dans un ensemble....136
Vérifier si un ensemble contient une valeur....136
Effectuer des opérations sur des ensembles....136
Ajout de valeurs de mon propre type à un ensemble....137
CountedSet....138
Chapitre 23: Enums....139
Remarques....139
Examples....139
Énumérations de base....139
Enums avec des valeurs associées....140
Charges indirectes....141
Valeurs brutes et hachages....141
Initialiseurs....142
Les énumérations partagent de nombreuses fonctionnalités avec les classes et les structure....143
Enumérations imbriquées....144
Chapitre 24: Fermetures....146
Syntaxe....146
Remarques....146

Examples....146
Bases de fermeture....146
Variations de syntaxe....147
Passer des fermetures dans des fonctions....147
Syntaxe de clôture....148
Paramètres @noescape....148
Swift 3 note:....149
throws et rethrows....149
Captures, références fortes / faibles et cycles de conservation....150
Conserver les cycles....150
Utilisation de fermetures pour le codage asynchrone....151
Fermetures et alias de type....152
Chapitre 25: Fonctionne comme des citoyens de première classe à Swift....153
Introduction....153
Examples....153
Affectation d'une fonction à une variable....153
Passer la fonction comme argument à une autre fonction, créant ainsi une fonction d'ordre....154
Fonction comme type de retour d'une autre fonction....154
Chapitre 26: Fonctions Swift Advance....155
Introduction....155
Examples....155
Introduction aux fonctions avancées....155
Aplatir un tableau multidimensionnel....156
Chapitre 27: Générer UIImage d'Initiales à partir de String....158
Introduction....158
Examples....158
InitialsImageFactory....158
Chapitre 28: Génériques....159
Remarques....159
Examples....159
Restriction des types d'espaces génériques....159
Les bases des génériques....160

Fonctions génériques 160
Types génériques 160
Passage de types génériques 161
Nom générique d'espace réservé 161
Exemples de classes génériques 161
Héritage de classe générique 162
Utilisation de génériques pour simplifier les fonctions de tableau 163
Utiliser des génériques pour améliorer la sécurité des types 163
Contraintes de type avancées 164
Chapitre 29: Gestion de la mémoire 166
Introduction 166
Remarques 166
Quand utiliser le mot-clé faible: 166
Quand utiliser le mot-clé sans propriétaire: 166
Pièges 166
Examples 166
Cycles de référence et références faibles 166
Références faibles 167
Gestion de la mémoire manuelle 168
Chapitre 30: Gestionnaire d'achèvement 169
Introduction 169
Examples 169
Gestionnaire d'achèvement sans argument d'entrée 169
Gestionnaire d'achèvement avec un argument d'entrée 169
Chapitre 31: Gestionnaire de paquets rapide 171
Examples 171
Création et utilisation d'un simple package Swift 171
Chapitre 32: Hachage cryptographique 173
Examples 173
MD2, MD4, MD5, SHA1, SHA224, SHA256, SHA384, SHA512 (Swift 3) 173
HMAC avec MD5, SHA1, SHA224, SHA256, SHA384, SHA512 (Swift 3) 174

Chapitre 33: Initialiseurs 177
Examples 177
Définition des valeurs de propriété par défaut 177
Personnalisation de l'initialisation avec les paramètres 177
Commencement init 178
Initialiseur désigné 181
Commodité init () 181
Init initiale (otherString: String) 181
Initialisateur désigné (appellera l'initialiseur désigné de la superclasse) 182
Commodité init () 182
Lanceur Jetable 182
Chapitre 34: Injection de dépendance 183
Examples 183
Injection de dépendance avec les contrôleurs de vue 183
Injection Dependenct Intro 183
Exemple sans DI 183
Exemple avec injection de dépendance 184
Types d'injection de dépendance 187
Exemple d'installation sans DI 187
Injection de dépendance d'initialisation 188
Propriétés Injection de dépendance 188
Méthode d'injection de dépendance 188
Chapitre 35: La déclaration différée 190
Examples 190
Quand utiliser une déclaration différée 190
Quand ne pas utiliser une déclaration différée 190
Chapitre 36: La gestion des erreurs 192
Remarques 192
Examples 192
Bases de traitement des erreurs 192
Attraper différents types d'erreur 193

Modèle de capture et de commutation pour la gestion explicite des erreurs 194
Désactivation de la propagation des erreurs 195
Créer une erreur personnalisée avec une description localisée 195
Chapitre 37: Lecture et écriture JSON 197
Syntaxe 197
Examples 197
Sérialisation JSON, encodage et décodage avec Apple Foundation et la bibliothèque Swift St 197
Lire JSON 197
Ecrire JSON 197
Encoder et décoder automatiquement 198
Encoder les données JSON 199
Décoder à partir de données JSON 199
Encodage ou décodage exclusivement 199
Utilisation de noms de clé personnalisés 199
SwiftyJSON 200
Freddy 201
Exemple de données JSON 202
Désérialisation des données brutes 202
Désérialisation des modèles directement 203
Sérialisation des données brutes 203
Sérialisation de modèles directement 203
Flèche 204
Analyse JSON simple dans des objets personnalisés 205
JSON Parsing Swift 3 206
Chapitre 38: Les extensions 209
Remarques 209
Examples 209
Variables et fonctions 209
Initialiseurs dans les extensions 209
Que sont les extensions? 210
Extensions de protocole 210
Restrictions 210

Que sont les extensions et quand les utiliser 211
Des indices 211
Chapitre 39: Les fonctions 213
Examples 213
Utilisation de base 213
Fonctions avec paramètres 213
Valeurs de retour 214
Erreurs de lancement 214
Les méthodes 215
Méthodes d'instance 215
Type Méthodes 215
Paramètres Inout 215
Syntaxe de fermeture de fuite 216
Les opérateurs sont des fonctions 216
Paramètres Variadic 216
Des indices 217
Options des indices: 218
Fonctions avec fermetures 218
Fonctions de passage et de retour 219
Types de fonctions 220
Chapitre 40: Méthode Swizzling 221
Remarques 221
Liens 221
Examples 221
Extension de UIViewController et Swizzling viewDidLoad 221
Bases de Swift Swizzling 222
Bases de Swizzling - Objective-C 223
Chapitre 41: Mise en cache sur l'espace disque 224
Introduction 224
Examples 224
Économie 224
En train de lire 224

Chapitre 42: Nombres 225
Examples 225
Types de nombres et littéraux 225
Littéraux 225
Entier la syntaxe littérale 225
Syntaxe littérale à virgule flottante 225
Convertir un type numérique en un autre 226
Convertir des nombres vers / depuis des chaînes 226
Arrondi 227
rond 227
plafond 227
sol 227
Int 228
Remarques 228
Génération de nombres aléatoires 228
Remarques 228
Exponentiation 229
Chapitre 43: NSRegularExpression dans Swift 230
Remarques 230
Examples 230
Extension de la chaîne pour faire correspondre un modèle simple 230
Utilisation de base 231
Remplacement des sous-couches 232
Caractères spéciaux 232
Validation 232
NSRegularExpression pour la validation du courrier 233
Chapitre 44: Objets associés 234
Examples 234
Propriété, dans une extension de protocole, obtenue à l'aide d'un objet associé 234
Chapitre 45: Opérateurs avancés 237
Examples 237

Opérateurs personnalisés....237
Surcharge + pour les dictionnaires....238
Opérateurs de communication....238
Opérateurs sur les bits....239
Opérateurs de débordement....240
Préséance des opérateurs Swift standard....240
Chapitre 46: Options....242
Introduction....242
Syntaxe....242
Remarques....242
Examples....242
Types d'option....242
Déballer une option....243
Opérateur de coalescence néant....244
Chaînage en option....244
Vue d'ensemble - Pourquoi les options?....245
Chapitre 47: OptionSet....247
Examples....247
Protocole OptionSet....247
Chapitre 48: Performance....248
Examples....248
Performance d'allocation....248
Avertissement sur les structures avec des chaînes et des propriétés qui sont des classes....248
Chapitre 49: Premiers pas avec la programmation orientée protocole....250
Remarques....250
Examples....250
Utilisation de la programmation orientée protocole pour les tests unitaires....250
Utilisation de protocoles comme types de première classe....251
Chapitre 50: Programmation fonctionnelle dans Swift....255
Examples....255
Extraire une liste de noms d'une liste de personne (s)....255
Traversée....255

En saillie.....255
Filtration.....256
Utiliser un filtre avec Structs.....257
Chapitre 51: Protocoles.....259
Introduction.....259
Remarques.....259
Examples.....259
Principes de base du protocole.....259
À propos des protocoles.....259
Exigences de type associé.....261
Modèle de délégué.....263
Extension de protocole pour une classe conforme spécifique.....264
Utilisation du protocole RawRepresentable (Extensible Enum).....265
Protocoles de classe uniquement.....266
Sémantique de référence des protocoles de classe uniquement.....266
Variables faibles de type de protocole.....267
Implémentation du protocole Hashable.....267
Chapitre 52: Réflexion.....269
Syntaxe.....269
Remarques.....269
Examples.....269
Utilisation de base pour miroir.....269
Obtenir le type et les noms des propriétés d'une classe sans avoir à l'instancier.....270
Chapitre 53: RxSwift.....273
Examples.....273
Bases RxSwift.....273
Créer des observables.....273
Jeter.....274
Fixations.....275
RxCocoa et ControlEvents.....275
Chapitre 54: Structs.....278
Examples.....278

Bases de Structs 278
Les structures sont des types de valeur 278
Mutant d'une structure 278
Lorsque vous pouvez utiliser des méthodes de mutation 279
Lorsque vous ne pouvez PAS utiliser de méthodes de mutation 279
Les structures ne peuvent pas hériter 279
Accéder aux membres de struct 279
Chapitre 55: Swift HTTP server par Kitura 281
Introduction 281
Examples 281
Bonjour application mondiale 281
Chapitre 56: Tableaux 285
Introduction 285
Syntaxe 285
Remarques 285
Examples 285
Sémantique de la valeur 285
Bases de tableaux 285
Tableaux vides 285
Littéraux de tableau 286
Tableaux avec valeurs répétées 286
Création de tableaux à partir d'autres séquences 286
Tableaux multidimensionnels 286
Accéder aux valeurs du tableau 287
Méthodes utiles 287
Modification de valeurs dans un tableau 288
Tri d'un tableau 288
Créer un nouveau tableau trié 288
Trier un tableau existant en place 288
Trier un tableau avec un ordre personnalisé 289
Transformer les éléments d'un tableau avec map (_ :) 289

Extraire des valeurs d'un type donné à partir d'un tableau avec flatMap (_ :) 290
Filtrage d'un tableau 290
Filtrage de la transformation Nil d'une matrice avec flatMap (_ :) 291
Inscrire un tableau avec une plage 291
Regroupement des valeurs de tableau 292
Aplatir le résultat d'une transformation Array avec flatMap (_ :) 292
Combiner les caractères dans un tableau de chaînes 293
Aplatir un tableau multidimensionnel 293
Trier un tableau de chaînes 293
Aplatir paresseusement un tableau multidimensionnel avec flatten () 294
Combiner les éléments d'un tableau avec réduire (_: combiner :) 295
Suppression d'un élément d'un tableau sans connaître son index 295
Swift3 295
Trouver l'élément minimum ou maximum d'un tableau 296
Trouver l'élément minimum ou maximum avec un ordre personnalisé 296
Accéder aux indices en toute sécurité 297
Comparaison de 2 tableaux avec zip 297
Chapitre 57: Travailler avec C et Objective-C 299
Remarques 299
Examples 299
Utiliser les classes Swift du code Objective-C 299
Dans le même module 299
Dans un autre module 300
Utilisation de classes Objective-C à partir du code Swift 300
Passerelles 300
Interface générée 301
Spécifiez un en-tête de pontage pour swiftc 302
Utiliser une carte de module pour importer des en-têtes C 302
Interaction fine entre Objective-C et Swift 303
Utilisez la bibliothèque standard C 304
Chapitre 58: Tuples 305
Introduction 305

Remarques 305
Examples 305
Que sont les tuples? 305
Se décomposer en variables individuelles 306
Tuples comme valeur de retour des fonctions 306
Utiliser un typealias pour nommer votre type de tuple 306
Echange de valeurs 307
Exemple avec 2 variables 307
Exemple avec 4 variables 307
Tuples comme cas dans Switch 308
Chapitre 59: Typealias 309
Examples 309
typealias pour les fermetures avec paramètres 309
typealias pour les fermetures vides 309
typealias pour d'autres types 309
Chapitre 60: Variables & Propriétés 310
Remarques 310
Examples 310
Créer une variable 310
Notions de base de la propriété 310
Propriétés stockées paresseuses 311
Propriétés calculées 311
Variables locales et globales 312
Propriétés du type 312
Observateurs de propriété 313
Crédits 315

À propos

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

Chapitre 1: Démarrer avec Swift Language

Remarques

Swift est un langage de programmation d'applications et de systèmes développé par Apple et distribué en tant que source ouverte . Swift interagit avec les API tactiles Objective-C et Cocoa / Cocoa pour les systèmes d'exploitation macOS, iOS, tvOS et watchOS d'Apple. Swift prend actuellement en charge macOS et Linux. Des efforts communautaires sont en cours pour prendre en charge Android, Windows et d’autres plates-formes.

Un développement rapide se produit sur GitHub ; les contributions sont généralement soumises via des demandes de tirage .

Les bogues et autres problèmes sont suivis sur bugs.swift.org .

Des discussions sur le développement, l’évolution et l’utilisation de Swift sont disponibles sur les listes de diffusion Swift .

Autres ressources

- Swift (langage de programmation) (Wikipedia)
- Le langage de programmation rapide (en ligne)
- Swift Standard Library Reference (en ligne)
- Directives de conception d'API (en ligne)
- Swift Programming Series (iBooks)
- ... et plus sur developer.apple.com .

Versions

Version rapide	Version Xcode	Date de sortie
le développement a commencé (premier engagement)	-	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
débuts open-source	-	2015-12-03

Version rapide	Version Xcode	Date de sortie
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

Votre premier programme Swift

Ecrivez votre code dans un fichier nommé `hello.swift` :

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

- Pour compiler et exécuter un script en une seule étape, utilisez `swift` depuis le terminal (dans un répertoire où se trouve ce fichier):

  Pour lancer un terminal, appuyez sur `CTRL` + `ALT` + `T` sous *Linux* ou trouvez-le dans Launchpad sur *macOS* . Pour changer de répertoire, entrez `cd directory_name` (ou `cd ..` pour revenir en arrière)

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

  Un **compilateur** est un programme informatique (ou un ensemble de programmes) qui transforme le code source écrit dans un langage de programmation (la langue source) en un autre langage informatique (la langue cible), cette dernière ayant souvent une forme binaire appelée code objet. (Wikipedia)

- Pour compiler et exécuter séparément, utilisez `swiftc` :

```
$ swiftc hello.swift
```

Cela compilera votre code dans un fichier `hello` . Pour l'exécuter, entrez `./` , suivi d'un nom de fichier.

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

- Ou utilisez le swift REPL (Read-Eval-Print-Loop), en tapant `swift` depuis la ligne de commande, puis en entrant votre code dans l'interpréteur:

Code:

```
func greet(name: String, surname: String) {
    print("Greetings \(name) \(surname)")
}

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

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

Divisons ce gros code en morceaux:

- `func greet(name: String, surname: String) { // function body }` - crée une *fonction* qui prend un `name` et un `surname` .
- `print("Greetings \(name) \(surname)")` - Imprime dans la console "Salutations", puis `name` , puis `surname` . Fondamentalement, `\(` **`variable_name`** `)` affiche la valeur de cette variable.
- `let myName = "Homer"` et `let mySurname = "Simpson"` - créer des *constantes* (variables dont la valeur ne peut pas changer) en utilisant `let` avec les noms: `myName` , `mySurname` et les valeurs: `"Homer"` , `"Simpson"` respectivement.
- `greet(name: myName, surname: mySurname)` - appelle une *fonction* que nous avons créée précédemment en fournissant les valeurs des *constantes* `myName` , `mySurname` .

En cours d'exécution en utilisant 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
```

Appuyez sur `CTRL` + `D` pour quitter REPL.

Installer Swift

Tout d'abord, téléchargez le compilateur et les composants.

Ensuite, ajoutez Swift à votre chemin. Sur macOS, l'emplacement par défaut de la chaîne d'outils téléchargeable est / Library / Developer / Toolchains. Exécutez la commande suivante dans Terminal:

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

Sous Linux, vous devrez installer clang:

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

Si vous avez installé la chaîne d'outils Swift dans un répertoire autre que la racine du système, vous devez exécuter la commande suivante, en utilisant le chemin d'accès réel de votre installation Swift:

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

Vous pouvez vérifier que vous avez la version actuelle de Swift en exécutant cette commande:

```
$ swift --version
```

Votre premier programme dans Swift sur un Mac (en utilisant un terrain de jeu)

Depuis votre Mac, téléchargez et installez Xcode sur le Mac App Store en suivant ce lien .

Une fois l'installation terminée, ouvrez Xcode et sélectionnez **Démarrer avec un terrain de jeu** :

Sur le panneau suivant, vous pouvez donner un nom à votre terrain de jeu ou vous pouvez le laisser `MyPlayground` et appuyer sur **Suivant** :

Choose options for your new playground:

Name MyPlayground

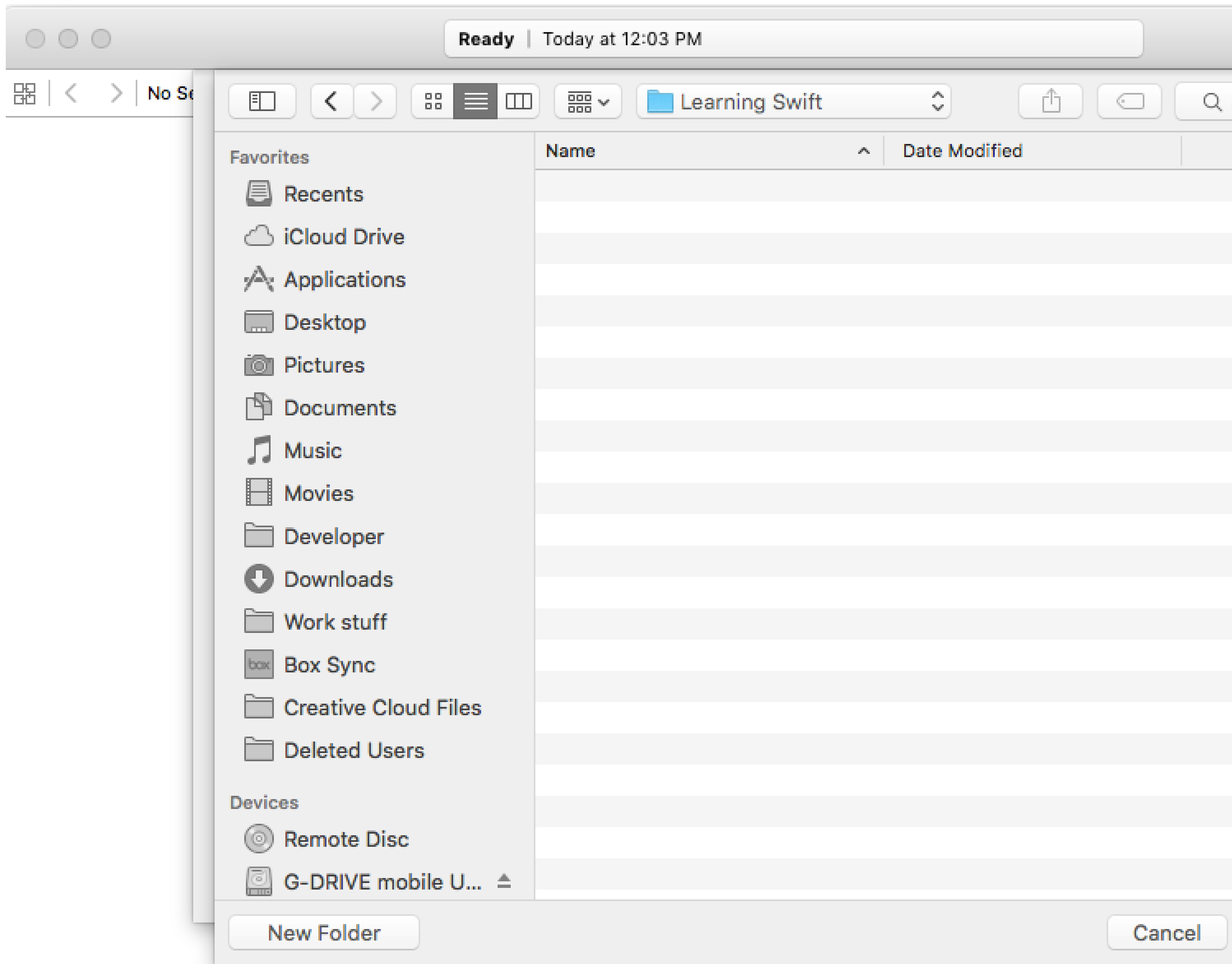
Platform: iOS

Cancel

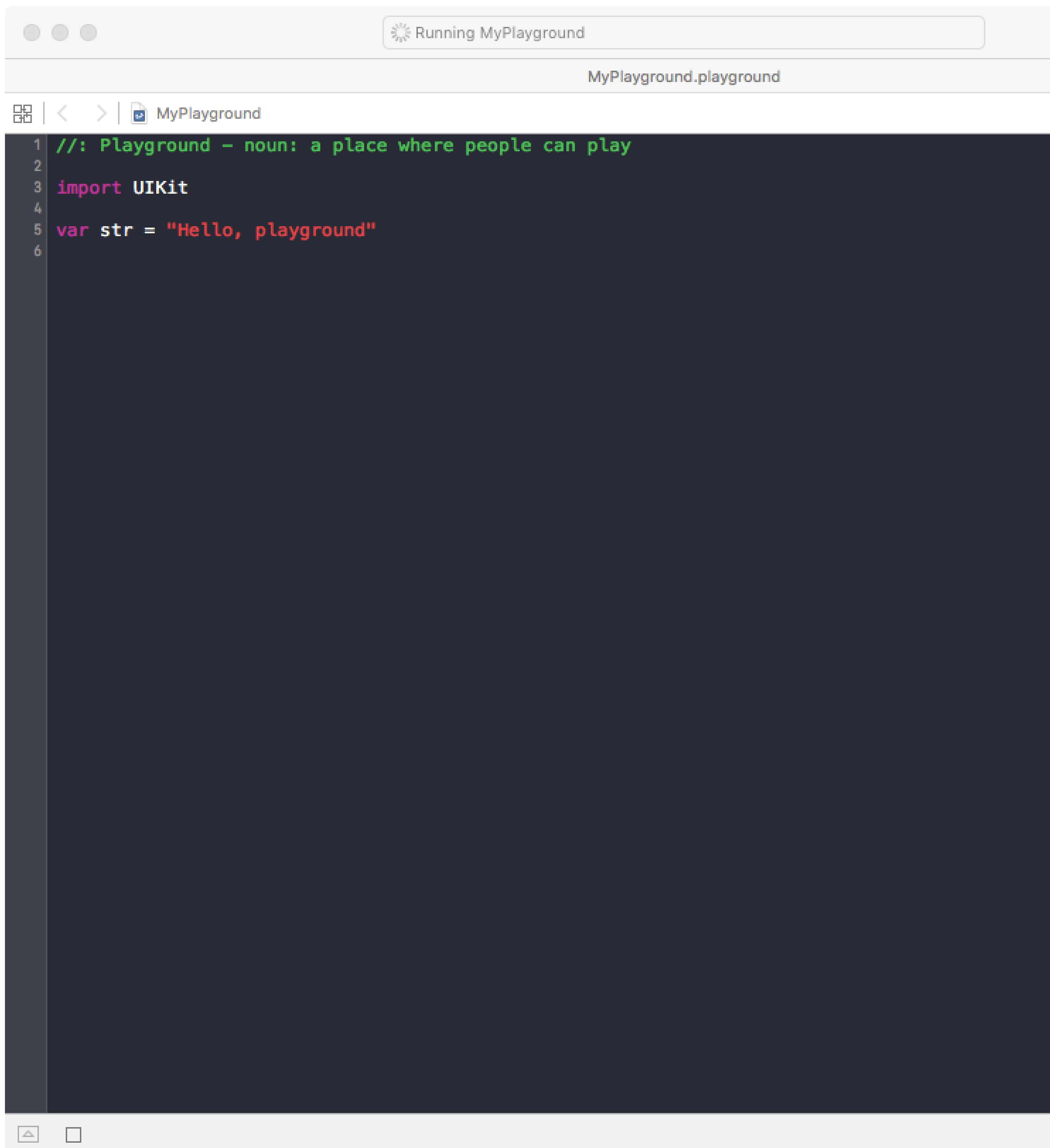
Previous

Next

Sélectionnez un endroit où enregistrer le terrain de jeu et appuyez sur **Créer** :



Le terrain de jeu s'ouvrira et votre écran devrait ressembler à ceci:



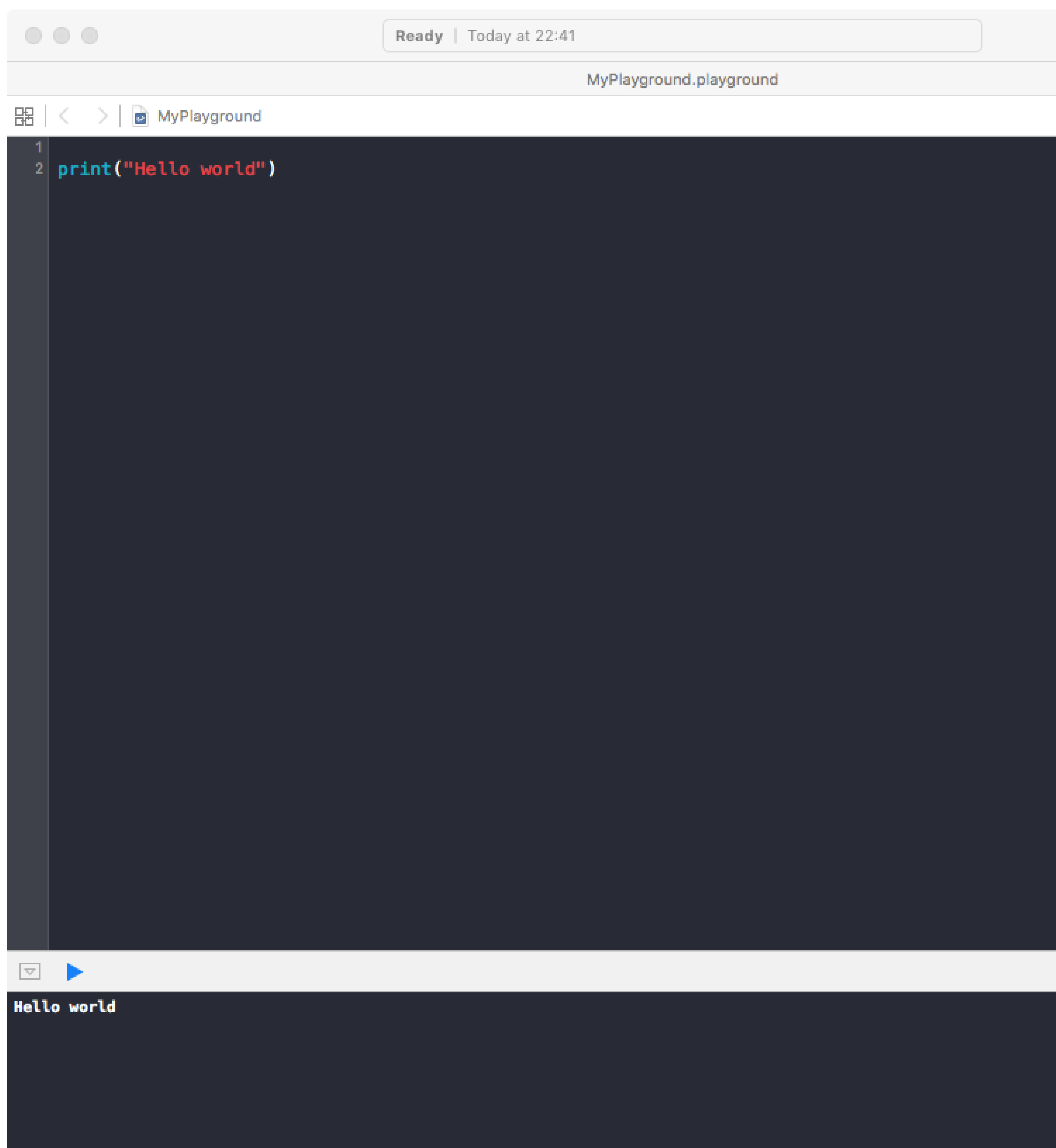
Maintenant que l'aire de jeu est à l'écran, appuyez sur + cmd + Y pour afficher la **zone** de **débogage.**

Enfin, supprimez le texte dans Playground et tapez:

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

Vous devriez voir "Hello world" dans la **zone de débogage** et "Hello world \ n" dans la **barre**

latérale droite:



Toutes nos félicitations! Vous avez créé votre premier programme dans Swift!

Votre premier programme dans l'application Swift Playgrounds sur iPad

L'application Swift Playgrounds est un excellent moyen de commencer à coder Swift en déplacement. Pour l'utiliser:

1- Téléchargez Swift Playgrounds pour iPad depuis App Store.

Mac iPad

iTunes Preview

Swift Playground

By Apple

Open iTunes to buy and do

View in iTunes

Free

2- Ouvrez l'application

3- Dans l'onglet **Mes terrains de jeu** , appuyez sur **+**

dans le coin supérieur gauche, puis sélectionnez Modèle vierge.

4- Entrez votre code.

5- Appuyez sur Exécuter mon code pour exécuter votre code.

6- A l'avant de chaque ligne, le résultat sera stocké dans un petit carré. Touchez-le pour révéler le résultat.

7- Pour parcourir le code lentement pour le tracer, appuyez sur le bouton à côté de Exécuter mon code.

Valeur optionnelle et énumération facultative

Type d'options, qui gère l'absence d'une valeur. Les optionnels disent soit "il y a une valeur, et c'est égal à x" ou "il n'y a pas de valeur du tout".

Un optionnel est un type en soi, en fait l'un des nouveaux enums super-puissants de Swift. Il a deux valeurs possibles, `None` et `Some(T)` , où T est une valeur associée du type de données correct disponible dans Swift.

Regardons par exemple ce morceau de code:

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

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

En fait, si vous ajoutez une **`print(x.dynamicType)`** dans le code ci-dessus, vous verrez ceci dans la console:

```
Optional<String>
```

Chaîne? est en fait du sucre syntaxique pour facultatif, et facultatif est un type à part entière.

Voici une version simplifiée de l'en-tête de Optional, que vous pouvez voir en cliquant sur le mot "Facultatif" dans votre code à partir de Xcode:

```
enum Optional<Wrapped> {

 /// The absence of a value.
 case none

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

Facultatif est en fait une énumération, définie par rapport à un type générique Wrapped. Il existe deux cas: *`.none`* pour représenter l'absence d'une valeur et *`.some`* pour représenter la présence d'une valeur, qui est stockée en tant que valeur associée de type Wrapped.

Laisse moi passer à nouveau: `String?` n'est pas une `String` mais une `Optional<String>` . Le fait que `Optional` soit un type signifie qu'il a ses propres méthodes, par exemple `map` et `flatMap` .

Lire Démarrer avec Swift Language en ligne: https://riptutorial.com/fr/swift/topic/202/demarrer-avec-swift-language

Chapitre 2: (Unsafe) Buffer Pointers

Introduction

“Un pointeur de tampon est utilisé pour un accès de bas niveau à une région de la mémoire. Par exemple, vous pouvez utiliser un pointeur de tampon pour un traitement et une communication efficaces des données entre les applications et les services.

Extrait de: Apple Inc. «Utilisation de Swift avec Cocoa et Objective-C (édition Swift 3.1)». IBooks. https://itun.es/us/utTW7.l

Vous êtes responsable de la gestion du cycle de vie de toute mémoire avec laquelle vous travaillez via des pointeurs de tampons, pour éviter les fuites ou un comportement indéfini.

Remarques

Concepts étroitement alignés **requis** pour compléter sa compréhension des BufferPointers (Unsafe).

- MemoryLayout (*Disposition de la mémoire d'un type, décrivant sa taille, sa foulée et son alignement* .)
- Unmanaged (*Type pour la propagation d'une référence d'objet non gérée*)
- UnsafeBufferPointer (*Une interface de collection non propriétaire à un tampon d'éléments stockés de manière contiguë dans la mémoire*).
- UnsafeBufferPointerIterator (*itérateur pour les éléments du tampon référencé par une occurrence UnsafeBufferPointer ou UnsafeMutableBufferPointer*)
- UnsafeMutableBufferPointer (*Une interface de collection non propriétaire à une mémoire tampon d'éléments mutables stockés de manière contiguë dans la mémoire*) *.*
- UnsafeMutablePointer (*Un pointeur pour accéder et manipuler des données d'un type spécifique*) *.*
- UnsafeMutableRawBufferPointer (*interface de collecte sans correspondance mutable aux octets dans une région de la mémoire.*)
- UnsafeMutableRawBufferPointer.Iterator (*Un itérateur sur les octets affichés par un pointeur de tampon brut.*)
- UnsafeMutableRawPointer (*Un pointeur brut pour accéder et manipuler des données non typées.*)
- UnsafePointer (*Un pointeur pour accéder aux données d'un type spécifique*)
- UnsafeRawBufferPointer (*Une interface de collecte non propriétaire aux octets dans une région de la mémoire.*)
- UnsafeRawBufferPointer.Iterator (*Un itérateur sur les octets affichés par un pointeur de tampon brut.*)
- UnsafeRawPointer (*Un pointeur brut pour accéder aux données non typées.*)

  (Source, Swiftdoc.org)

Examples

UnsafeMutablePointer

```
struct UnsafeMutablePointer<Pointee>
```

> Un pointeur pour accéder et manipuler des données d'un type spécifique.

Vous utilisez des instances du type UnsafeMutablePointer pour accéder aux données d'un type spécifique en mémoire. Le type de données auquel un pointeur peut accéder est le type Pointee du pointeur. UnsafeMutablePointer ne fournit aucune garantie de gestion de mémoire ou d'alignement automatique. Vous êtes responsable de la gestion du cycle de vie de toute mémoire avec laquelle vous travaillez via des pointeurs non sécurisés pour éviter les fuites ou un comportement indéfini.

La mémoire que vous gérez manuellement peut être non typée ou liée à un type spécifique. Vous utilisez le type UnsafeMutablePointer pour accéder et gérer la mémoire qui a été liée à un type spécifique. (Source)

```
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)
```

Converti à Swift 3.0 à partir de la source d' origine

Cas pratique d'utilisation des tampons

Déconstruire l'utilisation d'un pointeur non sécurisé dans la méthode de bibliothèque Swift;

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

Objectif:

Crée une nouvelle chaîne en copiant et en validant les données UTF-8 à terminaison nulle référencées par le pointeur donné.

Cet initialiseur ne tente pas de réparer les séquences d'unité de code UTF-8 mal formées. S'il en existe, le résultat de l'initialiseur est `nil` . L'exemple suivant appelle cet initialiseur avec des pointeurs vers le contenu de deux baies `CChar` différentes `CChar` la première avec des séquences d'unités de code UTF-8 bien formées et la seconde avec une séquence mal formée à la fin.

> **Source** , **Apple Inc., fichier d'en-tête Swift 3** (Pour l'accès en-tête: In Playground, Cmd + Clic sur le mot Swift) dans la ligne de code:
>
> `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"
```

> (Source, Apple Inc., Exemple de fichier Swift Header)

Lire (Unsafe) Buffer Pointers en ligne: https://riptutorial.com/fr/swift/topic/9140/-unsafe--buffer-pointers

Chapitre 3: Algorithmes avec Swift

Introduction

Les algorithmes sont l'épine dorsale de l'informatique. Faire un choix de l'algorithme à utiliser dans quelle situation distingue une moyenne d'un bon programmeur. Dans cet esprit, voici des définitions et des exemples de code de certains des algorithmes de base disponibles.

Examples

Tri par insertion

Le tri par insertion est l'un des algorithmes les plus élémentaires en informatique. Le tri par insertion classe les éléments en effectuant une itération dans une collection et positionne les éléments en fonction de leur valeur. L'ensemble est divisé en deux parties triées et non triées et se répète jusqu'à ce que tous les éléments soient triés. Le tri par insertion a une complexité de O (n2). Vous pouvez le mettre dans une extension, comme dans un exemple ci-dessous, ou vous pouvez créer une méthode pour cela.

```
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
}
}
```

Tri

Tri à bulles

Il s'agit d'un algorithme de tri simple qui passe en revue de manière répétée la liste à trier, compare chaque paire d'éléments adjacents et les échange s'ils sont dans le mauvais ordre. Le passage dans la liste est répété jusqu'à ce qu'aucun échange ne soit nécessaire. Bien que l'algorithme soit simple, il est trop lent et peu pratique pour la plupart des problèmes. Il a la complexité de O (n2) mais il est considéré comme plus lent que le tri par insertion.

```
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
}

}
```

Tri par insertion

Le tri par insertion est l'un des algorithmes les plus élémentaires en informatique. Le tri par insertion classe les éléments en effectuant une itération dans une collection et positionne les éléments en fonction de leur valeur. L'ensemble est divisé en deux parties triées et non triées et se répète jusqu'à ce que tous les éléments soient triés. Le tri par insertion a une complexité de O (n2). Vous pouvez le mettre dans une extension, comme dans un exemple ci-dessous, ou vous pouvez créer une méthode pour cela.

```
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
}
}
```

Tri de sélection

Le tri de sélection est noté pour sa simplicité. Il commence par le premier élément du tableau, en sauvegardant sa valeur en tant que valeur minimale (ou maximale, en fonction de l'ordre de tri). Il itère ensuite à travers le tableau et remplace la valeur min par toute autre valeur inférieure à min qu'elle trouve sur le chemin. Cette valeur min est alors placée à l'extrême gauche du tableau et le processus est répété, à partir de l'index suivant, jusqu'à la fin du tableau. Le tri par sélection a une complexité de O (n2) mais il est considéré comme plus lent que son homologue - Tri par sélection.

func selectionSort () -> Array {// vérifie la casse triviale 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
}
```

Tri rapide - O (n log n) temps de complexité

Quicksort est l'un des algorithmes avancés. Il présente une complexité temporelle de O (n log n) et applique une stratégie de division et de conquête. Cette combinaison se traduit par des performances algorithmiques avancées. Quicksort commence par diviser un grand tableau en deux sous-tableaux plus petits: les éléments bas et les éléments hauts. Quicksort peut ensuite trier récursivement les sous-tableaux.

Les étapes sont les suivantes:

Choisissez un élément, appelé pivot, dans le tableau.

Réorganisez le tableau de manière à ce que tous les éléments dont les valeurs sont inférieures au pivot arrivent avant le pivot, alors que tous les éléments dont les valeurs sont supérieures au pivot le suivent (les valeurs égales peuvent aller dans les deux sens). Après ce partitionnement, le pivot est dans sa position finale. Cela s'appelle l'opération de partition.

Appliquez récursivement les étapes ci-dessus au sous-tableau d'éléments avec des valeurs plus petites et séparément au sous-tableau d'éléments avec des valeurs supérieures.

func mutant 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
}
```

Tri de sélection

Le tri de sélection est noté pour sa simplicité. Il commence par le premier élément du tableau, en sauvegardant sa valeur en tant que valeur minimale (ou maximale, en fonction de l'ordre de tri). Il itère ensuite à travers le tableau et remplace la valeur min par toute autre valeur inférieure à min qu'elle trouve sur le chemin. Cette valeur min est alors placée à l'extrême gauche du tableau et le processus est répété, à partir de l'index suivant, jusqu'à la fin du tableau. Le tri par sélection a une complexité de O (n2) mais il est considéré comme plus lent que son homologue - Tri par sélection.

```
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
}
```

Analyse asymptotique

Comme nous avons le choix entre de nombreux algorithmes, lorsque nous voulons trier un tableau, nous devons savoir lequel fera son travail. Nous avons donc besoin d'une méthode pour mesurer la vitesse et la fiabilité de l'algoritm. C'est là qu'intervient l'analyse asymptotique. L'analyse asymptotique consiste à décrire l'efficacité des algorithmes au fur et à mesure que leur taille d'entrée (n) augmente. En informatique, les asymptotiques sont généralement exprimés dans un format commun appelé Big O Notation.

- **Temps linéaire O (n)** : Lorsque chaque élément du tableau doit être évalué pour qu'une fonction atteigne son objectif, cela signifie que la fonction devient moins efficace à mesure que le nombre d'éléments augmente. *Une fonction comme celle-ci est dite fonctionner en temps linéaire car sa vitesse dépend de sa taille d'entrée.*
- **Temps polynominal O (n2)** : si la complexité d'une fonction augmente de manière exponentielle (c'est-à-dire que pour n éléments d'une complexité de tableau d'une fonction est n au carré), cette fonction opère en temps polynominal. Ce sont généralement des fonctions avec des boucles imbriquées. Deux boucles imbriquées entraînent une complexité O (n2), trois boucles imbriquées entraînent une complexité O (n3), etc.
- **Temps logarithmique O (log n):** la complexité des fonctions de temps logarithmiques est minimisée lorsque la taille de ses entrées (n) augmente. Ce sont les types de fonctions que tout programmeur recherche.

Tri rapide - O (n log n) temps de complexité

Quicksort est l'un des algorithmes avancés. Il présente une complexité temporelle de O (n log n) et applique une stratégie de division et de conquête. Cette combinaison se traduit par des performances algorithmiques avancées. Quicksort commence par diviser un grand tableau en deux sous-tableaux plus petits: les éléments bas et les éléments hauts. Quicksort peut ensuite trier récursivement les sous-tableaux.

Les étapes sont les suivantes:

1. Choisissez un élément, appelé pivot, dans le tableau.

2. Réorganisez le tableau de manière à ce que tous les éléments dont les valeurs sont inférieures au pivot arrivent avant le pivot, alors que tous les éléments dont les valeurs sont supérieures au pivot le suivent (les valeurs égales peuvent aller dans les deux sens). Après ce partitionnement, le pivot est dans sa position finale. Cela s'appelle l'opération de partition.

3. Appliquez récursivement les étapes ci-dessus au sous-tableau d'éléments avec des valeurs plus petites et séparément au sous-tableau d'éléments avec des valeurs supérieures.

```
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
```

}

funcation mutante 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
}
```

Graph, Trie, Stack

Graphique

En informatique, un graphe est un type de données abstrait destiné à implémenter le graphe non orienté et les concepts de graphe orienté des mathématiques. Une structure de données de graphe consiste en un ensemble fini (et éventuellement mutable) de sommets ou de nœuds ou de points, ainsi qu’un ensemble de paires non ordonnées de ces sommets pour un graphe non orienté ou un ensemble de paires ordonnées pour un graphe orienté. Ces paires sont appelées arêtes, arcs ou lignes pour un graphe non orienté et comme flèches, arêtes dirigées, arcs orientés ou lignes dirigées pour un graphe orienté. Les sommets peuvent faire partie de la structure du graphe ou peuvent être des entités externes représentées par des index ou des références entiers. Une structure de données de graphique peut également associer à chaque arête une valeur de bord, telle qu'une étiquette symbolique ou un attribut numérique (coût, capacité, longueur, etc.). (Wikipedia, source)

```
//
//  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 informatique, un trie, également appelé arborescence numérique et parfois arborescence radix ou préfixe (comme on peut le rechercher par préfixes), est une sorte d'arbre de recherche - une structure de données d'arbre ordonnée utilisée pour stocker un ensemble dynamique ou associatif. tableau où les clés sont généralement des chaînes. (Wikipedia, source)

```
//
//  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, source)

Empiler

En informatique, une pile est un type de données abstrait qui sert de collection d'éléments, avec deux opérations principales: push, qui ajoute un élément à la collection, et pop, qui supprime l'élément le plus récemment ajouté qui n'a pas encore été supprimé. L'ordre dans lequel les éléments sortent d'une pile donne son autre nom, LIFO (pour last in, first out). De plus, une opération de coup d’œil peut donner accès au sommet sans modifier la pile. (Wikipedia, source)

Voir les informations de licence ci-dessous et le code source d'origine à (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 licence MIT (MIT)

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

Par la présente, toute personne obtenant une copie de ce logiciel et des fichiers de documentation associés (le «Logiciel»), sans restriction, y compris, sans limitation, les droits d’utilisation, de copie, de modification, de fusion, est autorisée. , publier, distribuer, concéder en sous-licence et / ou vendre des copies du logiciel, et autoriser les personnes à qui le logiciel est fourni à le faire, sous réserve des conditions suivantes:

L'avis de copyright ci-dessus et cet avis de permission doivent être inclus dans toutes les copies ou parties substantielles du logiciel.

LE LOGICIEL EST FOURNI "EN L'ÉTAT", SANS GARANTIE D'AUCUNE SORTE, EXPRESSE OU IMPLICITE, Y COMPRIS, MAIS SANS S'Y LIMITER, LES GARANTIES DE QUALITÉ MARCHANDE, D'ADÉQUATION À UN USAGE PARTICULIER ET DE NON-CONTREFAÇON. EN AUCUN CAS, LES AUTEURS OU LES TITULAIRES DE DROITS D'AUTEUR NE POURRONT ÊTRE TENUS RESPONSABLES D'UNE RÉCLAMATION, DOMMAGES OU AUTRE RESPONSABILITÉ, QUE CE SOIT PAR UN CONTRAT, UN TORT OU AUTRE, DÉCOULANT, HORS OU EN RELATION AVEC LE LOGICIEL OU LOGICIEL.

Lire Algorithmes avec Swift en ligne: https://riptutorial.com/fr/swift/topic/9116/algorithmes-avec-swift

Chapitre 4: Balisage de la documentation

Examples

Documentation de classe

Voici un exemple de documentation de classe de base:

```
/// 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
    }
}
```

Notez qu'avec *Xcode 8* , vous pouvez générer l'extrait de documentation avec la `commande` + `option` + `/` .

Cela va retourner:

Declaration	func say(content: String) -> Bool
Description	Method description
Parameters	content parameter description
Returns	return value description
Declared In	ViewController.swift

Styles de documentation

```
/**
 Adds user to the list of people 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` param
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

```
/**
 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
111
```

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

Lire Balisage de la documentation en ligne: https://riptutorial.com/fr/swift/topic/6937/balisage-de-la-documentation

Chapitre 5: Blocs

Introduction

De Swift Documentarion

Une fermeture est dite échapper à une fonction lorsque la fermeture est passée en argument à la fonction, mais est appelée après le retour de la fonction. Lorsque vous déclarez une fonction qui prend une fermeture comme l'un de ses paramètres, vous pouvez écrire @escaping avant le type du paramètre pour indiquer que la fermeture est autorisée à s'échapper.

Examples

Fermeture sans fuite

Dans Swift 1 et 2, les paramètres de fermeture s'échappaient par défaut. Si vous saviez que votre fermeture n'échapperait pas au corps de la fonction, vous pourriez marquer le paramètre avec l'attribut @noescape.

Dans Swift 3, c'est l'inverse: les paramètres de fermeture sont non-échappant par défaut. Si vous souhaitez que la fonction échappe à la fonction, vous devez la marquer avec l'attribut @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)
    }
  }
}
```

Fermeture de fermeture

De Swift Documentarion

> @ esquiver
>
> Appliquez cet attribut à un type de paramètre dans une déclaration de méthode ou de fonction pour indiquer que la valeur du paramètre peut être stockée pour une exécution

ultérieure. Cela signifie que la valeur est autorisée à survivre à la durée de vie de l'appel. Les paramètres de type de fonction avec l'attribut de type d'échappement nécessitent une utilisation explicite de self. pour les propriétés ou les méthodes.

```
class ClassThree {

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

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

Dans l'exemple ci-dessus, le bloc d'achèvement est enregistré à la fermeture et passera littéralement au-delà de l'appel de fonction. Donc complier va forcer à marquer le bloc d'achèvement comme @escaping.

Lire Blocs en ligne: https://riptutorial.com/fr/swift/topic/8623/blocs

Chapitre 6: Booléens

Examples

Qu'est-ce que Bool?

Bool est un type booléen avec deux valeurs possibles: `true` et `false` .

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

Les Bools sont utilisés dans les instructions de flux de contrôle en tant que conditions. L' instruction `if` utilise une condition booléenne pour déterminer le bloc de code à exécuter:

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

Annuler un Bool avec le préfixe! opérateur

Le préfixe `!` L'opérateur renvoie la **négation logique** de son argument. C'est-à-dire que `!true` renvoie `false` et `!false` renvoie `true` .

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

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

Opérateurs logiques booléens

L'opérateur OR (||) renvoie true si l'un de ses deux opérandes est évalué à true, sinon il renvoie false. Par exemple, le code suivant a la valeur true car au moins une des expressions de chaque côté de l'opérateur OR est true:

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

L'opérateur AND (&&) ne renvoie true que si les deux opérandes sont considérés comme vrais.

L'exemple suivant renvoie false car une seule des deux expressions d'opérande a la valeur true:

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

L'opérateur XOR (^) renvoie true si un et un seul des deux opérandes est évalué à true. Par exemple, le code suivant retournera true car un seul opérateur est considéré comme vrai:

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

Booléens et conditionnels en ligne

Une manière propre de gérer les booléens utilise un conditionnel en ligne avec le a? b: opérateur ternaire, qui fait partie des opérations de base de Swift.

Le conditionnel en ligne est composé de 3 composants:

```
question ? answerIfTrue : answerIfFalse
```

où question est un booléen évalué et answerIfTrue la valeur renvoyée si la question est vraie, et answerIfFalse la valeur renvoyée si la question est fausse.

L'expression ci-dessus est la même que:

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

Avec les conditionnels en ligne, vous retournez une valeur basée sur un booléen:

```
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'
```

Vous pouvez également appeler des méthodes basées sur une valeur booléenne:

```
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'
```

Les conditionnels en ligne permettent des évaluations booléennes à une seule ligne

Lire Booléens en ligne: https://riptutorial.com/fr/swift/topic/735/booleens

Chapitre 7: Boucles

Syntaxe

- pour constante en séquence {instructions}
- pour constante en séquence où condition {instructions}
- pour var variable en séquence {instructions}
- pour _ en séquence {instructions}
- pour le cas laisser constante dans la séquence {instructions}
- pour la casse constante dans l'ordre où condition {déclarations}
- pour la variable var dans la séquence {instructions}
- while condition {déclarations}
- répétez les {instructions} pendant que la condition est
- sequence.forEach (body: (Element) lance -> Void)

Examples

Boucle d'introduction

La boucle **for-in** vous permet d'effectuer une itération sur n'importe quelle séquence.

Itérer sur une gamme

Vous pouvez parcourir à la fois les plages ouvertes et fermées:

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

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

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

Itérer sur un tableau ou un ensemble

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

for name in names {
   print(name)
}

// James
```

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

2.1 2.2

Si vous avez besoin de l'index pour chaque élément du tableau, vous pouvez utiliser la méthode `enumerate()` sur `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()` renvoie une séquence paresseuse contenant des paires d'éléments avec `Int` s consécutives, à partir de 0. Par conséquent, avec les tableaux, ces nombres correspondent à l'index donné de chaque élément - mais cela peut ne pas être le cas avec d'autres types de collections.

3.0

Dans Swift 3, `enumerate()` a été renommé en `enumerated()` :

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

Itérer sur un dictionnaire

```
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.
```

Itérer en sens inverse

2.1 2.2

Vous pouvez utiliser la méthode `reverse()` sur `SequenceType` pour parcourir une séquence en sens inverse:

```
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

Dans Swift 3, `reverse()` a été renommé en `reversed()` :

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

Itérer sur des plages avec une foulée personnalisée

2.1 2.2

En utilisant les méthodes `stride(_:_:)` sur `Strideable` vous pouvez parcourir une plage avec une foulée personnalisée:

```
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

Dans Swift 3, les méthodes `stride(_:_:)` sur `Stridable` ont été remplacées par les fonctions `stride(_:_:_:)` globales `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)
}
```

Boucle répétée

Semblable à la boucle while, seule l'instruction de contrôle est évaluée après la boucle. Par conséquent, la boucle sera toujours exécutée au moins une fois.

```
var i: Int = 0

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

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

en boucle

A `while` boucle exécutera tant que la condition est vraie.

```
var count = 1

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

Type de séquence pour chaque bloc

Un type conforme au protocole SequenceType peut parcourir ses éléments dans une fermeture:

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

La même chose pourrait être faite avec un paramètre nommé:

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

* Remarque: Les instructions de flux de contrôle (telles que la rupture ou la poursuite) peuvent ne pas être utilisées dans ces blocs. Un retour peut être appelé, et s'il est appelé, il retournera immédiatement le bloc pour l'itération en cours (un peu comme un continu). La prochaine itération sera alors exécutée.

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

arr.forEach {

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

Boucle d'introduction avec filtrage

1. **`where`** clause

En ajoutant une clause `where` , vous pouvez restreindre les itérations à celles qui satisfont à la condition donnée.

```
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. clause de **`case`**

Il est utile de ne parcourir que les valeurs correspondant à un motif:

```
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
```

Vous pouvez également filtrer les valeurs facultatives et les déballer si nécessaire en ajoutant `?` marque après liaison constante:

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

```
//31
//5
```

Briser une boucle

Une boucle s'exécutera tant que sa condition reste vraie, mais vous pouvez l'arrêter manuellement à l'aide du mot-clé **`break`** . Par exemple:

```
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)
```

Lire Boucles en ligne: https://riptutorial.com/fr/swift/topic/1186/boucles

Chapitre 8: Casting de type

Syntaxe

- `let name = json["name"] as? String ?? ""` // Sortie: 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

Une variable peut être réduite à un sous-type en utilisant les *opérateurs* `as?` *type cast* `as?` et `as!` .

Le `as?` l'opérateur *tente* de convertir en sous-type.
Il peut échouer, donc il retourne une option.

```
let value: Any = "John"

let name = value as? String
print(name) // prints Optional("John")

let age = value as? Double
print(age) // prints nil
```

Le `as!` l'opérateur *force* un casting.
Il ne renvoie pas une option, mais se bloque si la distribution échoue.

```
let value: Any = "Paul"

let name = value as! String
print(name) // prints "Paul"

let age = value as! Double // crash: "Could not cast value…"
```

Il est courant d'utiliser des opérateurs de type cast avec un dépliage conditionnel:

```
let value: Any = "George"

if let name = value as? String {
    print(name) // prints "George"
}

if let age = value as? Double {
```

```
    print(age) // Not executed
}
```

Coulée avec interrupteur

L'instruction `switch` peut également être utilisée pour tenter de convertir en différents types:

```
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

L'opérateur `as` jettera un supertype. Comme il ne peut pas échouer, il ne retourne pas une option.

```
let name = "Ringo"
let value = string as Any  // `value` is of type `Any` now
```

Exemple d'utilisation d'un downcast sur un paramètre de fonction impliquant un sous-classement

Un downcast peut être utilisé pour utiliser le code et les données d'une sous-classe à l'intérieur d'une fonction en prenant un paramètre de sa super-classe.

```
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))
```

Type casting en langue rapide

Casting de type

Le transtypage est un moyen de vérifier le type d'une instance ou de traiter cette instance comme une super-classe ou une sous-classe différente de quelque part dans sa propre hiérarchie de classes.

Le type casting dans Swift est implémenté avec les opérateurs is et as. Ces deux opérateurs fournissent un moyen simple et expressif de vérifier le type d'une valeur ou de convertir une valeur en un type différent.

Downcasting

Une constante ou une variable d'un certain type de classe peut en fait se référer à une instance d'une sous-classe en coulisse. Lorsque vous pensez que cela est le cas, vous pouvez essayer de réduire à un type de sous-classe avec un opérateur de type cast (comme? Ou comme!).

Le downcasting pouvant échouer, l'opérateur de type cast se présente sous deux formes différentes. La forme conditionnelle, comme?, Renvoie une valeur facultative du type que vous essayez de réduire à. La forme forcée, as !, tente la diffusion descendante et force le défilement du résultat en une seule action composée.

Utilisez la forme conditionnelle de l'opérateur de type cast (comme?) Lorsque vous n'êtes pas sûr que le downcast réussira. Cette forme de l'opérateur renverra toujours une valeur facultative et la valeur sera nulle si le downcast n'était pas possible. Cela vous permet de vérifier la réussite d'un downcast.

Utilisez la forme forcée de l'opérateur de type cast (comme!) Uniquement lorsque vous êtes sûr que le downcast réussira toujours. Cette forme de l'opérateur déclenchera une erreur d'exécution si vous essayez de réduire à un type de classe incorrect. Savoir plus.

String to Int & Float conversion: -

```
    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
```

Float to String Conversion

```
    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"
```

Entier à valeur de chaîne

```
    let numbers = 888
    let intValue = String(numbers)
    print(intValue) // Output : "888"
```

Valeur flottante en chaîne

```
    let numbers = 888.00
    let floatValue = String(numbers)
    print(floatValue)
```

Valeur flottante facultative pour la chaîne

```
    let numbers: Any = 888.00
    let floatValue = String(describing: numbers)
    print(floatValue) // Output : 888.0
```

Chaîne facultative à la valeur Int

```
    let hitCount = "100"
    let data :AnyObject = hitCount
    let score = Int(data as? String ?? "") ?? 0
    print(score)
```

Valeurs de downcasting 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"]
```

Valeurs de downcasting de JSON facultatif

```
    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"]
```

Gérer la réponse JSON avec des conditions

```
    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"]
```

Gérer la réponse Nil avec condition

```
    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)
```

Sortie: `Empty Dictionary`

Lire Casting de type en ligne: https://riptutorial.com/fr/swift/topic/3082/casting-de-type

Chapitre 9: Chaînes et caractères

Syntaxe

- String.characters // Retourne un tableau des caractères de la chaîne
- String.characters.count // Retourne le nombre de caractères
- String.utf8 // Une String.UTF8View renvoie les points de caractère UTF-8 dans la chaîne.
- String.utf16 // Une String.UTF16View, renvoie les points de caractère UTF-16 dans la chaîne.
- String.unicodeScalars // Un String.UnicodeScalarView, renvoie les points de caractère UTF-32 dans la chaîne
- String.isEmpty // Renvoie true si la chaîne ne contient aucun texte
- String.hasPrefix (String) // Renvoie true si la chaîne est préfixée par l'argument
- String.hasSuffix (String) // Renvoie true si la chaîne est suffixée avec l'argument
- String.startIndex // Renvoie l'index correspondant au premier caractère de la chaîne
- String.endIndex // Retourne l'index qui correspond au spot après le dernier caractère de la chaîne
- String.components (sépare: String) // Retourne un tableau contenant les sous-chaînes séparées par la chaîne de séparation donnée
- String.append (Character) // Ajoute le caractère (donné en argument) à la chaîne

Remarques

Une `String` dans Swift est une collection de caractères et, par extension, une collection de scalaires Unicode. Comme les chaînes Swift sont basées sur Unicode, elles peuvent être n'importe quelle valeur scalaire Unicode, y compris les langues autres que l'anglais et les émoticônes.

Étant donné que deux scalaires peuvent être combinés pour former un seul caractère, le nombre de scalaires dans une chaîne n'est pas nécessairement le même que le nombre de caractères.

Pour plus d'informations sur les chaînes, voir Le langage de programmation Swift et la référence à la structure de chaîne .

Pour plus d'informations sur l'implémentation, voir "Swift String Design"

Examples

Littérature et caractères

Les littéraux de chaîne dans Swift sont délimités par des guillemets (`"`):

```
let greeting = "Hello!"  // greeting's type is String
```

Les caractères peuvent être initialisés à partir de littéraux de chaîne, à condition que le littéral ne contienne qu’un seul cluster de graphèmes:

```
let chr: Character = "H" // valid
let chr2: Character = "□" // valid
let chr3: Character = "abc" // invalid - multiple grapheme clusters
```

Interpolation de chaîne

L'interpolation de chaînes permet d'injecter une expression directement dans un littéral de chaîne. Cela peut être fait avec tous les types de valeurs, y compris les chaînes, les entiers, les nombres à virgule flottante et plus encore.

La syntaxe est une barre oblique inverse suivie de parenthèses qui entourent la valeur: `\(value)` . Toute expression valide peut apparaître entre parenthèses, y compris les appels de fonction.

```
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.
```

Pour les types personnalisés, le comportement par défaut de l'interpolation de chaîne est que `"\(myobj)"` est équivalent à `String(myobj)` , la même représentation utilisée par `print(myobj)` . Vous pouvez personnaliser ce comportement en implémentant le protocole `CustomStringConvertible` pour votre type.

3.0

Pour Swift 3, conformément à SE- `String.init<T>(_:)` , `String.init<T>(_:)` a été renommé `String.init<T>(describing:)` .

L'interpolation de chaîne `"\(myobj)"` préfèrera l' `String.init<T: LosslessStringConvertible>(_:)` , mais retombera sur `init<T>(describing:)` si la valeur n'est pas `LosslessStringConvertible` .

Caractères spéciaux

Certains caractères nécessitent une **séquence d'échappement** spéciale pour les utiliser dans les littéraux de chaîne:

Personnage	Sens
`\0`	le caractère nul
`\\`	un simple backslash, `\`

Personnage	Sens
`\t`	un caractère de tabulation
`\v`	un onglet vertical
`\r`	un retour de chariot
`\n`	un saut de ligne ("newline")
`\"`	une double citation, `"`
`\'`	une seule citation, `'`
`\u{n}`	le point de code Unicode *n* (en hexadécimal)

Exemple:

```
let message = "Then he said, \"I \u{1F496} you!\""

print(message) // Then he said, "I 💖 you!"
```

Concaténer des chaînes

Concaténer des chaînes avec l'opérateur `+` pour produire une nouvelle chaîne:

```
let name = "John"
let surname = "Appleseed"
let fullName = name + " " + surname  // fullName is "John Appleseed"
```

Ajouter à une chaîne **mutable** à l'aide de l' opérateur d'attribution composé `+=` ou en utilisant une méthode:

```
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"
```

Ajouter un seul caractère à une chaîne modifiable:

```
var greeting: String = "Hello"
let exclamationMark: Character = "!"
greeting.append(exclamationMark)
// produces a modified String (greeting) = "Hello!"
```

Ajouter plusieurs caractères à une chaîne modifiable

```
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(_:)` a été renommé pour `append(_:)` .

Joignez une séquence de chaînes pour former une nouvelle chaîne à l'aide de `joinWithSeparator(_:)` :

```
let words = ["apple", "orange", "banana"]
let str = words.joinWithSeparator(" & ")

print(str)   // "apple & orange & banana"
```

3.0

`joinWithSeparator(_:)` a été renommé en `joinWithSeparator(_:)` `joined(separator:)` .

Le `separator` est la chaîne vide par défaut, donc `["a", "b", "c"].joined() == "abc"` .

Examinez et comparez les chaînes

Vérifiez si une chaîne est vide:

```
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
```

Vérifiez si deux chaînes sont égales (au sens de l' équivalence canonique 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
```

Vérifiez si une chaîne commence / se termine par une autre chaîne:

```
"fortitude".hasPrefix("fort")      // true
"Swift Language".hasSuffix("age")  // true
```

Codage et décomposition de chaînes

Une chaîne Swift est constituée de points de code Unicode . Il peut être décomposé et encodé de différentes manières.

```
let str = "ที่👌①!"
```

Cordes de décomposition

Une chaîne de `characters` sont Unicode groupes de graphèmes étendus :

```
Array(str.characters)  // ["ที่", "👌", "①", "!"]
```

Les `unicodeScalars` sont les points de code Unicode qui constituent une chaîne (notez que ที่ est un groupe de graphèmes, mais que 3 points de code - 3607, 3637, 3656 - la longueur du tableau résultant est différente de celle des `characters`):

```
str.unicodeScalars.map{ $0.value }  // [3607, 3637, 3656, 128076, 9312, 33]
```

Vous pouvez encoder et décomposer des chaînes en UTF-8 (une séquence d' `UInt8` s) ou en UTF-16 (une séquence d' `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]
```

Longueur de la chaîne et itération

Une chaîne de `characters` , `unicodeScalars` , `utf8` et `utf16` sont tous Collection s, afin que vous puissiez obtenir leur `count` et itérer sur eux:

```
// 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

Réglage des valeurs

Utiliser Unicode directement

```
var str: String = "I want to visit □□, Москва, मुंबई, القاهرة, and □□□. □"
var character: Character = "□"
```

Utiliser des valeurs hexadécimales

```
var str: String = "\u{61}\u{5927}\u{1F34E}\u{3C0}" // a□□π
var character: Character = "\u{65}\u{301}" // é = "e" + accent mark
```

Notez que le `Character` rapide peut être composé de plusieurs points de code Unicode, mais semble être un seul caractère. C'est ce qu'on appelle un cluster Grapheme étendu.

Conversions

String -> Hex

```
// Accesses views of different Unicode encodings of `str`
str.utf8
str.utf16
str.unicodeScalars // UTF-32
```

Hex -> String

```
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‼□
```

Notez que pour UTF-8 et UTF-16, la conversion n'est pas toujours aussi simple car les choses comme les emoji ne peuvent pas être encodées avec une seule valeur UTF-16. Il faut une paire de substitution.

Cordes d'inversion

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)
```

Chaînes majuscules et minuscules

Pour que tous les caractères d'une chaîne soient en majuscule ou en minuscule:

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"
```

Vérifier si la chaîne contient des caractères d'un ensemble défini

Des lettres

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 nouvelle structure `CharacterSet` également reliée à la classe Objective-C `NSCharacterSet` définit plusieurs ensembles prédéfinis comme `NSCharacterSet` :

- `decimalDigits`
- `capitalizedLetters`
- `alphanumerics`
- `controlCharacters`
- `illegalCharacters`
- et plus vous pouvez trouver dans la référence NSCharacterSet .

Vous pouvez également définir votre propre jeu de caractères:

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")
}
```

Vous pouvez également inclure la plage:

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")
}
```

Compter les occurrences d'un personnage dans une chaîne

Étant donné une `String` et un `Character`

```
let text = "Hello World"
let char: Character = "o"
```

Nous pouvons compter le nombre de fois que le `Character` apparaît dans la `String` utilisant

```
let sensitiveCount = text.characters.filter { $0 == char }.count // case-sensitive
let insensitiveCount = text.lowercaseString.characters.filter { $0 ==
Character(String(char).lowercaseString) } // case-insensitive
```

Supprimer des caractères d'une chaîne non définie dans 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)
```

Chaînes de formatage

Zéros de tête

```
let number: Int = 7
let str1 = String(format: "%03d", number) // 007
let str2 = String(format: "%05d", number) // 00007
```

Nombres après décimal

```
let number: Float = 3.14159
let str1 = String(format: "%.2f", number) // 3.14
let str2 = String(format: "%.4f", number) // 3.1416 (rounded)
```

Décimal à hexadécimal

```
let number: Int = 13627
let str1 = String(format: "%2X", number) // 353B
let str2 = String(format: "%2x", number) // 353b (notice the lowercase b)
```

On peut aussi utiliser un initialiseur spécialisé qui fait la même chose:

```
let number: Int = 13627
let str1 = String(number, radix: 16, uppercase: true) //353B
let str2 = String(number, radix: 16) // 353b
```

Décimal à un nombre avec une base arbitraire

```
let number: Int = 13627
let str1 = String(number, radix: 36) // aij
```

Radix est `Int` dans `[2, 36]` .

Conversion d'une chaîne Swift en un type numérique

```
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
```

Notez que cela retourne une valeur `Optional` , qui doit être déballée en conséquence avant d'être utilisée.

Itération de chaîne

3.0

```
let string = "My fantastic string"
var index = string.startIndex

while index != string.endIndex {
    print(string[index])
    index = index.successor()
}
```

Remarque: `endIndex` situe après la fin de la chaîne (par exemple, `string[string.endIndex]` est une erreur, mais `string[string.startIndex]` est `string[string.startIndex]`). En outre, dans une chaîne vide (`""`), `string.startIndex == string.endIndex` est `true` . Assurez-vous de vérifier les chaînes vides, car vous ne pouvez pas appeler `startIndex.successor()` sur une chaîne vide.

3.0

Dans Swift 3, les index String n'ont plus `successor()` , `predecessor()` , `advancedBy(_:)` , `advancedBy(_:limit:)` ou `distanceTo(_:)` .

Au lieu de cela, ces opérations sont déplacées vers la collection, qui est maintenant responsable de l'incrémentation et de la décrémentation de ses index.

Les méthodes disponibles sont `.index(after:)` , `.index(before:)` et `.index(_:, offsetBy:)` .

```
let string = "My fantastic string"
var currentIndex = string.startIndex

while currentIndex != string.endIndex {
    print(string[currentIndex])
    currentIndex = string.index(after: currentIndex)
}
```

Note: nous utilisons `currentIndex` comme nom de variable pour éviter toute confusion avec la méthode `.index` .

Et, par exemple, si vous voulez aller dans l'autre sens:

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
    }
}
```

(Ou vous pouvez simplement inverser la chaîne en premier, mais si vous n'avez pas besoin de parcourir toute la chaîne, vous préférerez probablement une méthode comme celle-ci)

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
    }
}
```

Remarque: `Index` est un type d'objet, et non un `Int` . Vous ne pouvez pas accéder à un caractère de chaîne comme suit:

```
let string = "My string"
string[2] // can't do this
string.characters[2] // and also can't do this
```

Mais vous pouvez obtenir un index spécifique comme suit:

3.0

```
index = string.startIndex.advanceBy(2)
```

3.0

```
currentIndex = string.index(string.startIndex, offsetBy: 2)
```

Et peut revenir en arrière comme ceci:

3.0

```
index = string.endIndex.advancedBy(-2)
```

3.0

```
currentIndex = string.index(string.endIndex, offsetBy: -2)
```

Si vous pouvez dépasser les limites de la chaîne, ou si vous souhaitez spécifier une limite, vous pouvez utiliser:

3.0

```
index = string.startIndex.advanceBy(20, limit: string.endIndex)
```

3.0

```
currentIndex = string.index(string.startIndex, offsetBy: 20, limitedBy: string.endIndex)
```

Alternativement, on peut simplement parcourir les caractères d'une chaîne, mais cela peut être moins utile selon le contexte:

```
for c in string.characters {
    print(c)
}
```

Supprimer les WhiteSpace et NewLine de début et de fin

3.0

```
let someString = "  Swift Language  \n"
let trimmedString =
someString.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceAndNewlineCharacterSet())
// "Swift Language"
```

La méthode `stringByTrimmingCharactersInSet` renvoie une nouvelle chaîne créée en supprimant des deux extrémités des caractères String contenus dans un jeu de caractères donné.

Nous pouvons également simplement supprimer uniquement les espaces ou les nouvelles lignes.

Supprimer uniquement les espaces:

```
let trimmedWhiteSpace =
someString.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceCharacterSet())
// "Swift Language  \n"
```

Enlever uniquement la nouvelle ligne:

```
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  "
```

Note: toutes ces méthodes appartiennent à `Foundation` . Utilisez `import Foundation` si Foundation n'est pas déjà importé via d'autres bibliothèques comme Cocoa ou UIKit.

Convertir une chaîne depuis et vers Data / NSData

Pour convertir une chaîne vers et depuis Data / NSData, nous devons encoder cette chaîne avec un codage spécifique. Le plus connu est `UTF-8` une représentation à 8 bits de caractères Unicode, adaptée à la transmission ou au stockage par des systèmes ASCII. Voici une liste de tous les `String Encodings` disponibles

`String` de `Data` / `NSData`

3.0

```
let data = string.data(using: .utf8)
```

2.2

```
let data = string.dataUsingEncoding(NSUTF8StringEncoding)
```

`Data` / `NSData` en `String`

3.0

```
let string = String(data: data, encoding: .utf8)
```

2.2

```
let string = String(data: data, encoding: NSUTF8StringEncoding)
```

Fractionnement d'une chaîne en un tableau

Dans Swift, vous pouvez facilement séparer une chaîne en un tableau en le découpant à un certain caractère:

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"]`
```

Ou quand le séparateur n'est pas présent:

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"]
```

Lire Chaînes et caractères en ligne: https://riptutorial.com/fr/swift/topic/320/chaines-et-caracteres

Chapitre 10: Commutateur

Paramètres

Paramètre	Détails
Valeur à tester	La variable à comparer

Remarques

Fournissez un cas pour chaque valeur possible de votre saisie. Utilisez un `default case` pour couvrir les valeurs d'entrée restantes que vous ne souhaitez pas spécifier. Le cas par défaut doit être le dernier cas.

Par défaut, les commutateurs dans Swift ne continueront pas à vérifier d'autres cas après la correspondance d'un cas.

Examples

Utilisation de base

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

Les instructions switch fonctionnent également avec des types de données autres que des entiers. Ils fonctionnent avec n'importe quel type de données. Voici un exemple d'activation d'une chaîne:

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

Cela imprimera les éléments suivants:

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

Faire correspondre plusieurs valeurs

Un seul cas dans une instruction switch peut correspondre à plusieurs valeurs.

```
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")
}
```

Faire correspondre une plage

Un seul cas dans une instruction switch peut correspondre à une plage de valeurs.

```
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")
}
```

Utilisation de l'instruction where dans un commutateur

L'instruction where peut être utilisée dans une correspondance de casse de commutateur pour ajouter des critères supplémentaires requis pour une correspondance positive. L'exemple suivant vérifie non seulement la plage, mais aussi si le nombre est impair ou pair:

```
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")
}
```

Satisfaire l'une des contraintes multiples en utilisant switch

Vous pouvez créer un tuple et utiliser un commutateur comme ceci:

```
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")
}
```

Correspondance partielle

L'instruction switch utilise une correspondance partielle.

```
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. Correspond exactement au cas où la valeur est (0,0,0). C'est l'origine de l'espace 3D.
2. Correspond à y = 0, z = 0 et toute valeur de x. Cela signifie que la coordonnée est sur l'axe des x.
3. Correspond à x = 0, z = 0 et à toute valeur de y. Cela signifie que la coordonnée est sur leur axe.
4. Correspond à x = 0, y = 0 et à toute valeur de z. Cela signifie que la coordonnée est sur l'axe des z.
5. Correspond au reste des coordonnées.

Remarque: l'utilisation du trait de soulignement signifie que vous ne vous souciez pas de la valeur.

Si vous ne voulez pas ignorer la valeur, vous pouvez l'utiliser dans votre instruction switch, comme ceci:

```
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)")
}
```

Ici, les cas des axes utilisent la syntaxe let pour extraire les valeurs pertinentes. Le code imprime ensuite les valeurs en utilisant une interpolation de chaîne pour générer la chaîne.

Remarque: vous n'avez pas besoin d'un paramètre par défaut dans cette instruction de commutateur. C'est parce que le cas final est essentiellement le défaut - il correspond à tout, car il n'y a aucune contrainte sur une partie quelconque du tuple. Si l'instruction switch change toutes les valeurs possibles avec ses cases, aucune valeur par défaut n'est nécessaire.

Nous pouvons également utiliser la syntaxe let-where pour rechercher des cas plus complexes. Par exemple:

```
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
}
```

Ici, nous faisons correspondre les lignes "y est égal à x" et "y est égal à x carré".

Changement de vitesse

Il est intéressant de noter que, rapidement, contrairement à d'autres langages, les gens sont familiers, il y a une rupture implicite à la fin de chaque déclaration de cas. Afin de suivre le cas suivant (c.-à- `fallthrough` . `fallthrough` plusieurs cas `fallthrough`), vous devez utiliser `fallthrough` déclaration de `fallthrough` .

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

Ceci est utile pour des choses comme les flux.

Switch et Enums

L'instruction Switch fonctionne très bien avec les valeurs 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")
}
```

> Comme nous avons fourni un cas pour chaque valeur possible de voiture, nous omettons le cas `default` .

Commutateur et options

Quelques exemples de cas où le résultat est facultatif.

```
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")
}
```

Commutateurs et tuples

Les commutateurs peuvent activer les tuples:

```
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)"
}
```

Correspondance basée sur la classe - idéal pour prepareForSegue

Vous pouvez également faire un changement d'instruction de commutateur en fonction de la **classe** de la chose que vous allumez.

Un exemple où cela est utile est dans `prepareForSegue` . J'avais l'habitude de basculer en fonction de l'identifiant de segue, mais c'est fragile. Si vous modifiez votre storyboard ultérieurement et renommez l'identifiant de segue, il casse votre code. Ou, si vous utilisez segues pour plusieurs instances de la même classe de contrôleur de vue (mais pour différentes scènes de storyboard), vous ne pouvez pas utiliser l'identificateur de segue pour déterminer la classe de la destination.

Swift switch déclarations à la rescousse.

Utilisez la `case let var as Class` Swift `case let var as Class` syntaxe `case let var as Class` , comme ceci:

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

Dans Swift 3, le sytax a légèrement changé:

```
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
  }
}
```

Lire Commutateur en ligne: https://riptutorial.com/fr/swift/topic/207/commutateur

Chapitre 11: Concurrence

Syntaxe

- Swift 3.0
- DispatchQueue.main // Récupère la file d'attente principale
- DispatchQueue (label: "my-serial-queue", attributs: [.serial, .qosBackground]) // Crée votre propre file d'attente série privée
- DispatchQueue.global (attributs: [.qosDefault]) // Accéder à l'une des files d'attente simultanées globales
- DispatchQueue.main.async {...} // Envoie une tâche de manière asynchrone au thread principal
- DispatchQueue.main.sync {...} // Envoie une tâche de manière synchrone au thread principal
- DispatchQueue.main.asyncAfter (date limite: .now () + 3) {...} // Envoie une tâche de manière asynchrone au thread principal à exécuter après x secondes
- Swift <3.0
- dispatch_get_main_queue () // Récupère la file d'attente principale sur le thread principal
- dispatch_get_global_queue (dispatch_queue_priority_t, 0) // Récupère la file d'attente globale avec la priorité spécifiée dispatch_queue_priority_t
- dispatch_async (dispatch_queue_t) {() -> Void in ...} // Envoie une tâche de manière asynchrone sur le dispatch_queue_t spécifié
- dispatch_sync (dispatch_queue_t) {() -> Void in ...} // Envoie une tâche de manière synchrone sur le dispatch_queue_t spécifié
- dispatch_after (dispatch_time (DISPATCH_TIME_NOW, Int64 (nanosecondes)), dispatch_queue_t, {...}); // Envoie une tâche sur le dispatch_queue_t spécifié après des nanosecondes

Examples

Obtention d'une file d'attente Grand Central Dispatch (GCD)

Grand Central Dispatch travaille sur le concept de "files d'attente d'expédition". Une file d'attente de distribution exécute les tâches que vous avez désignées dans l'ordre dans lequel elles ont été passées. Il existe trois types de files d'attente:

- **Serial Dispatch Queues** (aka files d'attente de distribution privées) exécutent une tâche à la fois, dans l'ordre. Ils sont fréquemment utilisés pour synchroniser l'accès à une ressource.
- **Les files d'attente de répartition simultanées** (**c.-à-d** . Les files d'attente de répartition globales) exécutent une ou plusieurs tâches simultanément.
- La **file d'attente de répartition principale** exécute les tâches sur le thread principal.

Pour accéder à la file d'attente principale:

3.0

```
let mainQueue = DispatchQueue.main
```

3.0

```
let mainQueue = dispatch_get_main_queue()
```

Le système fournit *des* files d'attente de répartition mondiales *simultanées* (globales à votre application), avec des priorités variables. Vous pouvez accéder à ces files d'attente en utilisant la classe `DispatchQueue` dans Swift 3:

3.0

```
let globalConcurrentQueue = DispatchQueue.global(qos: .default)
```

équivalent à

```
let globalConcurrentQueue = DispatchQueue.global()
```

3.0

```
let globalConcurrentQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)
```

Dans iOS 8 ou version ultérieure, les valeurs de qualité de service pouvant être transmises sont les `.userInteractive` : `.userInitiated` , `.default` , `.utility` , `.background` et `.background` . Ceux-ci remplacent les constantes `DISPATCH_QUEUE_PRIORITY_` .

Vous pouvez également créer vos propres files d'attente avec différentes priorités:

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)
```

Dans Swift 3, les files d'attente créées avec cet initialiseur sont en série par défaut et le passage de `.workItem` à la fréquence d'autorelease garantit qu'un pool d'autorelease est créé et vidé pour chaque élément de travail. Il y a aussi `.never` , ce qui signifie que vous allez gérer vous-même vos pools d'autorelease, ou `.inherit` qui hérite des paramètres de l'environnement. Dans la plupart des cas, vous n'utiliserez probablement jamais `.never` sauf en cas de personnalisation extrême.

Exécution de tâches dans une file d'attente Grand Central Dispatch (GCD)

3.0

Pour exécuter des tâches sur une file d'attente de distribution, utilisez les méthodes `sync` , `async` et `after` .

Pour envoyer une tâche dans une file d'attente de manière asynchrone:

```
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
```

Pour envoyer une tâche dans une file d'attente de manière synchrone:

```
queue.sync {
    // Do some task
}
// ... code here will not execute until the task is finished
```

Pour envoyer une tâche dans une file d'attente après un certain nombre de secondes:

```
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
```

> **REMARQUE:** Toute mise à jour de l'interface utilisateur doit être appelée sur le thread principal! Assurez-vous de mettre le code des mises à jour d'interface utilisateur dans `DispatchQueue.main.async { ... }`

2.0

Types de file d'attente:

```
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)
```

Pour envoyer une tâche dans une file d'attente de manière asynchrone:

```
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
```

Pour envoyer une tâche dans une file d'attente de manière synchrone:

```
dispatch_sync(queue) {
    // Your sync code
}
// Code after the sync block will wait until the sync task finished
```

Pour envoyer une tâche après un intervalle de temps (utilisez `NSEC_PER_SEC` pour convertir les secondes en nanosecondes):

```
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
}
```

Pour exécuter une tâche de manière asynchrone et mettre à jour l'interface utilisateur:

```
dispatch_async(queue) {
    // Your time consuming code here
    dispatch_async(dispatch_get_main_queue()) {
        // Update the UI code
    }
}
```

> **REMARQUE:** Toute mise à jour de l'interface utilisateur doit être appelée sur le thread principal! Assurez-vous de mettre le code des mises à jour de l'interface utilisateur dans `dispatch_async(dispatch_get_main_queue()) { ... }`

Boucles simultanées

GCD fournit un mécanisme pour effectuer une boucle, les boucles se produisant simultanément les unes par rapport aux autres. Ceci est très utile lorsque vous effectuez une série de calculs coûteux.

Considérez cette boucle:

```
for index in 0 ..< iterations {
    // Do something computationally expensive here
}
```

Vous pouvez effectuer ces calculs simultanément à l'aide de `concurrentPerform` (dans Swift 3) ou de `dispatch_apply` (dans 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
}
```

La fermeture de la boucle sera appelée pour chaque `index` entre `0` et, mais sans inclure, les `iterations` . Ces itérations seront exécutées simultanément les unes par rapport aux autres et, par conséquent, l'ordre qu'elles exécutent n'est pas garanti. Le nombre réel d'itérations qui se produisent simultanément à un moment donné est généralement dicté par les capacités du périphérique en question (par exemple, le nombre de cœurs du périphérique).

Quelques considérations spéciales:

- `concurrentPerform` / `dispatch_apply` peut exécuter les boucles simultanément les unes par rapport aux autres, mais tout cela se produit de manière synchrone par rapport au thread à partir duquel vous l'appelez. Donc, n'appelez pas cela depuis le thread principal, car cela bloquera ce thread jusqu'à ce que la boucle soit terminée.
- Étant donné que ces boucles se produisent simultanément les unes par rapport aux autres, vous êtes responsable de la sécurité des threads. Par exemple, si vous mettez à jour un dictionnaire avec les résultats de ces calculs coûteux, veillez à synchroniser ces mises à jour vous-même.
- Notez qu'il existe des frais généraux associés à l'exécution de boucles concurrentes. Par conséquent, si les calculs effectués à l'intérieur de la boucle ne nécessitent pas beaucoup de calculs, vous pouvez constater que les performances obtenues en utilisant des boucles simultanées peuvent être diminuées, voire totalement compensées, par la surcharge associée à la synchronisation de tous ces threads.

  Vous êtes donc responsable de déterminer la quantité correcte de travail à effectuer dans chaque itération de la boucle. Si les calculs sont trop simples, vous pouvez utiliser "striding" pour inclure plus de travail par boucle. Par exemple, plutôt que de faire une boucle simultanée avec 1 million de calculs triviaux, vous pouvez effectuer 100 itérations dans votre boucle, en effectuant 10 000 calculs par boucle. De cette façon, il y a suffisamment de travail effectué sur chaque thread, de sorte que la surcharge associée à la gestion de ces boucles simultanées devient moins importante.

Exécution de tâches dans une OperationQueue

Vous pouvez penser à `OperationQueue` comme une ligne de tâches en attente d'exécution. Contrairement aux files d'attente d'expédition dans GCD, les files d'attente d'opérations ne sont pas des FIFO (premier entré, premier sorti). Au lieu de cela, ils exécutent des tâches dès qu'ils sont prêts à être exécutés, tant qu'il y a suffisamment de ressources système pour le permettre.

Obtenez le principal `OperationQueue` :

3.0

```
let mainQueue = OperationQueue.main
```

Créez une `OperationQueue` personnalisée:

3.0

```
let queue = OperationQueue()
queue.name = "My Queue"
queue.qualityOfService = .default
```

Qualité de service spécifie l’importance du travail ou la probabilité que l’utilisateur compte sur les résultats immédiats de la tâche.

Ajouter une `Operation` à une `OperationQueue` :

3.0

```
// An instance of some Operation subclass
let operation = BlockOperation {
    // perform task here
}

queue.addOperation(operation)
```

Ajouter un bloc à `OperationQueue` :

3.0

```
myQueue.addOperation {
    // some task
}
```

Ajoutez plusieurs `Operation` à une `OperationQueue` :

3.0

```
let operations = [Operation]()
// Fill array with Operations

myQueue.addOperation(operations)
```

Ajustez le nombre d' `Operation` pouvant être exécutées simultanément dans la file d'attente:

```
myQueue.maxConcurrentOperationCount = 3 // 3 operations may execute at once

// Sets number of concurrent operations based on current system conditions
myQueue.maxConcurrentOperationCount = NSOperationQueueDefaultMaxConcurrentOperationCount
```

La suspension d'une file d'attente l'empêchera de lancer l'exécution d'opérations existantes non démarrées ou de nouvelles opérations ajoutées à la file d'attente. La manière de reprendre cette file d'attente consiste à définir `isSuspended` sur `false` :

3.0

```
myQueue.isSuspended = true

// Re-enable execution
myQueue.isSuspended = false
```

La suspension d'une `OperationQueue` n'arrête pas ou n'annule pas les opérations en cours d'exécution. On ne doit tenter de suspendre une file d'attente que vous avez créée, pas les files d'attente globales ou la file d'attente principale.

Création d'opérations de haut niveau

Le framework Foundation fournit le type `Operation` , qui représente un objet de haut niveau qui encapsule une partie du travail pouvant être exécutée dans une file d'attente. Non seulement la file d'attente coordonne les performances de ces opérations, mais vous pouvez également établir des dépendances entre les opérations, créer des opérations annulables, limiter le degré de concurrence utilisé par la file d'attente d'opérations, etc.

`Operation` deviennent prêtes à être exécutées lorsque toutes ses dépendances sont terminées. La propriété `isReady` alors `true` .

Créez une sous-classe `Operation` simple non concurrente:

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
    }

}
```

Ajouter une opération à une `OperationQueue` :

1.0

```
myQueue.addOperation(operation)
```

Cela exécutera l'opération simultanément dans la file d'attente.

Gérer les dépendances sur une `Operation` .

Les dépendances définissent d'autres `Operation` qui doivent s'exécuter dans une file d'attente avant que l' `Operation` soit considérée prête à être exécutée.

1.0

```
operation2.addDependency(operation1)

operation2.removeDependency(operation1)
```

Exécutez une `Operation` sans file d'attente:

1.0

```
    operation.start()
```

Les dépendances seront ignorées. S'il s'agit d'une opération simultanée, la tâche peut toujours être exécutée simultanément si sa méthode de `start` décharge le travail sur les files d'attente en arrière-plan.

Opérations simultanées

Si la tâche que l' `Operation` consiste à effectuer est, lui - même, asynchrone, (par exemple , une `URLSession` tâche de données), vous devez mettre en œuvre l' `Operation` comme une opération concurrente. Dans ce cas, votre implémentation `isAsynchronous` doit retourner `true` , vous aurez généralement une méthode `start` qui effectue une configuration, puis appelle sa méthode `main` qui exécute réellement la tâche.

Lorsque vous implémentez une `Operation` asynchrone, vous devez implémenter les méthodes `isExecuting` , `isFinished` et KVO. Ainsi, lorsque l'exécution commence, la propriété `isExecuting` devient `true` . Lorsqu'une `Operation` termine sa tâche, `isExecuting` est défini sur `false` et `isFinished` est défini sur `true` . Si l'opération est annulée, `isCancelled` et `isFinished` `true` . Toutes ces propriétés sont observables par valeur-clé.

Annuler une `Operation` .

L'appel d' `cancel` simplement la propriété `isCancelled` sur `true` . Pour répondre à l'annulation depuis votre propre sous-classe `Operation` , vous devez vérifier la valeur de `isCancelled` au moins périodiquement dans `main` et répondre de manière appropriée.

1.0

```
operation.cancel()
```

Lire Concurrence en ligne: https://riptutorial.com/fr/swift/topic/1649/concurrence

Chapitre 12: Conditionnels

Introduction

Les expressions conditionnelles, impliquant des mots clés tels que if, else if et else, permettent aux programmes Swift d'effectuer différentes actions en fonction d'une condition booléenne: True ou False. Cette section traite de l'utilisation des conditionnels Swift, de la logique booléenne et des instructions ternaires.

Remarques

Pour plus d'informations sur les instructions conditionnelles, voir Le langage de programmation Swift .

Examples

Utiliser la garde

2.0

La garde vérifie une condition et si elle est fausse, elle pénètre dans la branche. Les agences de contrôle de garde doivent quitter leur bloc fermé soit par `return` , soit par `break` , soit par `continue` (le cas échéant); Si vous ne le faites pas, cela entraîne une erreur de compilation. Cela a l'avantage que lorsqu'un `guard` est écrit, il n'est pas possible de laisser le flux continuer accidentellement (comme cela serait possible avec un `if`).

L'utilisation de gardes peut aider à réduire les niveaux d'imbrication , ce qui améliore généralement la lisibilité du code.

```
func printNum(num: Int) {
    guard num == 10 else {
        print("num is not 10")
        return
    }
    print("num is 10")
}
```

Guard peut également vérifier s'il existe une valeur dans une option , puis la dérouler dans la portée externe:

```
func printOptionalNum(num: Int?) {
    guard let unwrappedNum = num else {
        print("num does not exist")
        return
    }
    print(unwrappedNum)
}
```

Guard peut combiner un désemballage et une vérification de condition en utilisant `where` mot-clé:

```
func printOptionalNum(num: Int?) {
guard let unwrappedNum = num, unwrappedNum == 10 else {
    print("num does not exist or is not 10")
    return
}
print(unwrappedNum)
}
```

Conditions élémentaires: instructions if

Une **instruction `if`** vérifie si une condition Bool est `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` instructions acceptent **`else if`** et **`else`** bloque, ce qui permet de tester des conditions alternatives et de fournir un repli:

```
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")
}
```

Opérateurs de base comme `&&` et `||` peut être utilisé pour plusieurs conditions:

L'opérateur logique

```
let num = 10
let str = "Hi"
if num == 10 && str == "Hi" {
    print("num is 10, AND str is \"Hi\"")
}
```

Si `num == 10` était faux, la deuxième valeur ne serait pas évaluée. Cela s'appelle l'évaluation des courts-circuits.

L'opérateur logique OU

```
if num == 10 || str == "Hi" {
    print("num is 10, or str is \"Hi\")
}
```

Si `num == 10` est vrai, la seconde valeur ne sera pas évaluée.

L'opérateur logique NOT

```
if !str.isEmpty {
    print("str is not empty")
}
```

Liaison facultative et clauses "where"

Les options doivent être *déballées* avant de pouvoir être utilisées dans la plupart des expressions. `if let` est une *liaison facultative* , qui réussit si la valeur facultative n'était **pas** **`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")
}
```

Vous pouvez réutiliser le **même nom** pour la variable nouvellement liée, en masquant l'original:

```
// num originally has type Int?
if let num = num {
    // num has type Int inside this block
}
```

1,2 3,0

Combinez plusieurs liaisons facultatives avec des virgules (`,`):

```
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
}
```

Appliquez d'autres contraintes après la liaison facultative en utilisant une clause `where` :

```
if let unwrappedNum = num where unwrappedNum % 2 == 0 {
    print("num is non-nil, and it's an even number")
}
```

Si vous vous sentez aventureux, entrelacez un nombre quelconque de liaisons optionnelles et de clauses `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

Dans Swift 3, `where` les clauses ont été remplacées (SE-0099): il suffit d' utiliser une autre `,` pour séparer les liaisons optionnelles et des conditions booléennes.

```
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!
}
```

Opérateur ternaire

Les conditions peuvent également être évaluées sur une seule ligne à l'aide de l'opérateur ternaire:

Si vous souhaitez déterminer le minimum et le maximum de deux variables, vous pouvez utiliser des instructions if, comme ceci:

```
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
}
```

L'opérateur conditionnel ternaire prend une condition et renvoie l'une des deux valeurs, selon

que la condition est vraie ou fausse. La syntaxe est la suivante: Cela équivaut à avoir l'expression:

```
(<CONDITION>) ? <TRUE VALUE> : <FALSE VALUE>
```

Le code ci-dessus peut être réécrit en utilisant un opérateur conditionnel ternaire comme ci-dessous:

```
let a = 5
let b = 10
let min = a < b ? a : b
let max = a > b ? a : b
```

Dans le premier exemple, la condition est un <b. Si cela est vrai, le résultat renvoyé à min sera de a; si c'est faux, le résultat sera la valeur de b.

Remarque: Comme il est courant de trouver le plus grand ou le plus petit des deux nombres, la bibliothèque standard Swift offre deux fonctions à cet effet: max et min.

Nil-Coalescing Operator

L'opérateur de coalescence nulle <OPTIONAL> ?? <DEFAULT VALUE> déballe <OPTIONAL> s'il contient une valeur ou renvoie <DEFAULT VALUE> si est nul. <OPTIONAL> est toujours d'un type facultatif. <DEFAULT VALUE> doit correspondre au type stocké dans <OPTIONAL> .

L'opérateur nuls-coalescing est un raccourci pour le code ci-dessous qui utilise un opérateur ternaire:

```
a != nil ? a! : b
```

ceci peut être vérifié par le code ci-dessous:

```
(a ?? b) == (a != nil ? a! : b) // ouputs true
```

Temps pour un exemple

```
let defaultSpeed:String = "Slow"
var userEnteredSpeed:String? = nil

print(userEnteredSpeed ?? defaultSpeed) // ouputs "Slow"

userEnteredSpeed = "Fast"
print(userEnteredSpeed ?? defaultSpeed) // ouputs "Fast"
```

Lire Conditionnels en ligne: https://riptutorial.com/fr/swift/topic/475/conditionnels

Chapitre 13: Contrôle d'accès

Syntaxe

- Projet de classe privée
- let car = Car ("Ford", modèle: "Escape") // par défaut interne
- enum public
- func calcaire à calculerMarketCap ()
- remplacer la configuration interne du func setupView ()
- private (set) var area = 0

Remarques

1. ***Remarque de base:***

Vous trouverez ci-dessous les trois niveaux d'accès du plus haut accès (le moins restrictif) au plus faible accès (le plus restrictif).

L' accès **public** permet d'accéder aux classes, aux structures, aux variables, etc. à partir de n'importe quel fichier du modèle, mais plus important encore en dehors du module si le fichier externe importe le module contenant le code d'accès public. Il est populaire d'utiliser l'accès public lorsque vous créez un cadre.

L' accès **interne** permet aux fichiers uniquement avec le module des entités d'utiliser les entités. Toutes les entités ont **un** niveau d'accès **interne** par défaut (à quelques exceptions près).

L' accès **privé** empêche l'entité d'être utilisée en dehors de ce fichier.

2. ***Sous-classement Remarque:***

Une sous-classe ne peut pas avoir un accès plus élevé que sa super-classe.

3. ***Getter & Setter Remarque:***

Si le setter de la propriété est privé, le getter est interne (qui est la valeur par défaut). Vous pouvez également attribuer un niveau d'accès à la fois au getter et au setter. Ces principes s'appliquent également aux *indices*

4. ***Remarque générale:***

Les autres types d'entité incluent: les initialiseurs, les protocoles, les extensions, les génériques et les alias de type.

Examples

Exemple de base utilisant une structure

3.0

Dans Swift 3, il y a plusieurs niveaux d'accès. Cet exemple les utilise tous sauf pour 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)"
    }
}
```

Supposons que myCar été initialisé comme ceci:

```
let myCar = Car("Apple", model: "iCar")
```

Car.make (public)

```
print(myCar.make)
```

Cette impression fonctionnera partout, y compris les cibles qui importent de la Car .

Car.model (interne)

```
print(myCar.model)
```

Cela compilera si le code est dans la même cible que Car .

Car.otherName (fichierprivate)

```
print(myCar.otherName)
```

Cela ne fonctionnera que si le code est *dans le même fichier* que Car .

Car.fullName (privé)

```
print(myCar.fullName)
```

Cela ne fonctionnera pas dans Swift 3. Les propriétés private ne sont accessibles que dans la même 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 l'entité a plusieurs niveaux d'accès associés, Swift recherche le niveau d'accès le plus bas. Si une variable privée existe dans une classe publique, la variable sera toujours considérée comme privée.

Exemple de sous-classement

```
public class SuperClass {
    private func secretMethod() {}
}

internal class SubClass: SuperClass {
    override internal func secretMethod() {
        super.secretMethod()
    }
}
```

Exemple de Getters et Setters

```
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() {}
}
```

Lire Contrôle d'accès en ligne: https://riptutorial.com/fr/swift/topic/1075/controle-d-acces

Chapitre 14: Conventions de style

Remarques

Swift a un guide de style officiel: Swift.org API Design Guidelines . Un autre guide populaire est le guide officiel Swift Style de raywenderlich.com.

Examples

Effacer l'utilisation

Éviter l'ambiguïté

Le nom des classes, structures, fonctions et variables doit éviter toute ambiguïté.

Exemple:

```
extension List {
    public mutating func remove(at position: Index) -> Element {
        // implementation
    }
}
```

L'appel de fonction à cette fonction ressemblera alors à ceci:

```
list.remove(at: 42)
```

De cette façon, l'ambiguïté est évitée. Si l'appel de fonction ne serait que list.remove(42) il serait difficile de savoir si un élément égal à 42 serait supprimé ou si l'élément de l'index 42 serait supprimé.

Éviter la redondance

Le nom des fonctions ne doit pas contenir d'informations redondantes.

Un mauvais exemple serait:

```
extension List {
    public mutating func removeElement(element: Element) -> Element? {
        // implementation
    }
}
```

Un appel à la fonction peut ressembler à list.removeElement(someObject) . La variable someObject indique déjà qu'un élément est supprimé. Il serait préférable que la signature de la fonction ressemble à ceci:

```
extension List {
    public mutating func remove(_ member: Element) -> Element? {
        // implementation
    }
}
```

L'appel à cette fonction ressemble à ceci: list.remove(someObject) .

Nommer les variables en fonction de leur rôle

Les variables doivent être nommées par leur rôle (par exemple fournisseur, message d'accueil) au lieu de leur type (par exemple, usine, chaîne, etc.)

Couplage élevé entre le nom du protocole et les noms de variables

Si le nom du type décrit son rôle dans la plupart des cas (par exemple Iterator), le type doit être nommé avec le suffixe `Type`. (par exemple, IteratorType)

Fournir des détails supplémentaires lors de l'utilisation de paramètres faiblement typés

Si le type d'un objet n'indique pas clairement son utilisation dans un appel de fonction, la fonction doit être nommée avec un nom précédent pour chaque paramètre faiblement typé, décrivant son utilisation.
Exemple:

```
func addObserver(_ observer: NSObject, forKeyPath path: String)
```

à quoi un appel ressemblerait `object.addObserver (self, forKeyPath: path)

au lieu de

```
func add(_ observer: NSObject, for keyPath: String)
```

à quoi un appel ressemblerait à object.add(self, for: path)

Usage Courant

Utiliser le langage naturel

Les appels de fonctions doivent être proches de la langue anglaise naturelle.

Exemple:

```
list.insert(element, at: index)
```

au lieu de

```
list.insert(element, position: index)
```

Méthodes de nommage

Les méthodes d'usine devraient commencer par le préfixe `make`.

Exemple:

```
factory.makeObject()
```

Paramètres de dénomination dans les initialiseurs et les méthodes d'usine

Le nom du premier argument ne doit pas être utilisé pour nommer une méthode de fabrique ou un initialiseur.

Exemple:

```
factory.makeObject(key: value)
```

Au lieu de:

```
factory.makeObject(havingProperty: value)
```

Nommer selon les effets secondaires

- Les fonctions avec effets secondaires (fonctions mutantes) doivent être nommées en utilisant des verbes ou des noms préfixés par form- .
- Fonctions sans effets secondaires (fonctions nonmutating) doivent être nommés en utilisant des noms ou des verbes avec le suffixe -ing ou -ed .

Exemple: Fonctions 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
```

Fonctions non-mutantes:

```
let sortedArray = array.sorted()     // out of place sorting
let union = set.union(anotherSet)    // union is now the union of set and another set
```

Fonctions booléennes ou variables

Les instructions impliquant des booléens doivent se lire comme des assertions.

Exemple:

```
set.isEmpty
line.intersects(anotherLine)
```

Protocoles de nommage

- Les protocoles décrivant quelque chose doivent être nommés à l'aide de noms.
- Les protocoles décrivant les capacités doivent avoir un suffixe -able , -ible ou -ing .

Exemple:

```
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
```

Types et propriétés

Les types, les variables et les propriétés doivent être considérés comme des noms.

Exemple:

```
let factory = ...
```

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

Capitalisation

Types et protocoles

Les noms de type et de protocole doivent commencer par une lettre majuscule.

Exemple:

```
protocol Collection {}
struct String {}
class UIView {}
struct Int {}
enum Color {}
```

Tout le reste...

Les variables, constantes, fonctions et énumérations doivent commencer par une lettre minuscule.

Exemple:

```
let greeting = "Hello"
let height = 42.0

enum Color {
    case red
    case green
    case blue
}

func print(_ string: String) {
    ...
}
```

Affaire de chameau:

Tous les noms doivent utiliser le cas de chameau approprié. Boîtier supérieur pour les noms de type / protocole et boitier inférieur pour tout le reste.

Upper Camel Case:

```
protocol IteratorType { ... }
```

Boitier inférieur:

```
let inputView = ...
```

Abréviations

Les abréviations doivent être évitées, sauf si elles sont couramment utilisées (par exemple, URL, ID). Si une abréviation est utilisée, toutes les lettres doivent avoir la même casse.

Exemple:

```
let userID: UserID = ...
let urlString: URLString = ...
```

Lire Conventions de style en ligne: https://riptutorial.com/fr/swift/topic/3031/conventions-de-style

Chapitre 15: Cryptage AES

Examples

Cryptage AES en mode CBC avec un IV aléatoire (Swift 3.0)

Le iv est préfixé aux données cryptées

aesCBC128Encrypt créera un IV aléatoire et préfixé au code chiffré.
aesCBC128Decrypt utilisera le préfixe IV lors du déchiffrement.

Les entrées sont les données et les clés sont des objets de données. Si un formulaire codé tel que Base64 si nécessaire, convertissez-le en et / ou de la méthode d'appel.

La clé doit comporter exactement 128 bits (16 octets), 192 bits (24 octets) ou 256 bits (32 octets). Si une autre taille de clé est utilisée, une erreur sera générée.

Le remplissage PKCS # 7 est défini par défaut.

Cet exemple nécessite Common Crypto
Il est nécessaire d'avoir un en-tête de pontage pour le projet:
#import <CommonCrypto/CommonCrypto.h>
Ajoutez le Security.framework au projet.

Ceci est un exemple, pas de code de production.

```
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;
}
```

Exemple d'utilisation:

```
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)")
}
```

Exemple de sortie:

```
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>
```

Remarques:
Un problème typique avec le code d'exemple de mode CBC est qu'il laisse la création et le partage du caractère aléatoire IV à l'utilisateur. Cet exemple inclut la génération de l'IV, préfixé les données cryptées et utilise le préfixe IV lors du décryptage. Cela libère l'utilisateur occasionnel des détails nécessaires au mode CBC .

Pour la sécurité, les données cryptées doivent également avoir une authentification, cet exemple de code ne le fournit pas pour être petit et permettre une meilleure interopérabilité pour les autres plates-formes.

Il manque également la dérivation de clé de la clé à partir d'un mot de passe, il est suggéré d'utiliser PBKDF2 lorsque les mots de passe de texte sont utilisés comme matériel de saisie.

Pour un code de chiffrement multiplateforme robuste et prêt pour la production, voir RNCryptor .

> Mis à jour pour utiliser des tailles de clé / de jet / multiple basées sur la clé fournie.

Cryptage AES en mode CBC avec un IV aléatoire (Swift 2.3)

Le iv est préfixé aux données cryptées

aesCBC128Encrypt créera un IV aléatoire et préfixé au code chiffré. aesCBC128Decrypt utilisera le préfixe IV lors du déchiffrement.

Les entrées sont les données et les clés sont des objets de données. Si un formulaire codé tel que Base64 si nécessaire, convertissez-le en et / ou de la méthode d'appel.

La clé doit être exactement de 128 bits (16 octets). Pour les autres tailles de clé, voir l'exemple Swift 3.0.

Le remplissage PKCS # 7 est défini par défaut.

Cet exemple nécessite Common Crypto Il est nécessaire d'avoir un en-tête de pontage pour le projet: #import <CommonCrypto / CommonCrypto.h> Ajoutez le fichier Security.framework au projet.

Voir Swift 3 exemple pour les notes.

Ceci est un exemple, pas de code de production.

```
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;
}
```

Exemple d'utilisation:

```
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))")
```

Exemple de sortie:

```
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>
```

Cryptage AES en mode ECB avec remplissage PKCS7

De la documentation Apple pour IV,

> Ce paramètre est ignoré si le mode ECB est utilisé ou si un algorithme de chiffrement de flux est sélectionné.

```
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
    }
```

Lire Cryptage AES en ligne: https://riptutorial.com/fr/swift/topic/7026/cryptage-aes

Chapitre 16: Dérivation de clé PBKDF2

Examples

Dérivation de clés par mot de passe 2 (Swift 3)

La dérivation de clé basée sur un mot de passe peut être utilisée à la fois pour dériver une clé de chiffrement du texte du mot de passe et pour enregistrer un mot de passe à des fins d'authentification.

Plusieurs algorithmes de hachage peuvent être utilisés, notamment SHA1, SHA256, SHA512, fournis par cet exemple de code.

Le paramètre rounds est utilisé pour ralentir le calcul afin qu'un attaquant passe beaucoup de temps à chaque tentative. Les valeurs de retard typiques se situent entre 100 ms et 500 ms, des valeurs plus courtes peuvent être utilisées si les performances sont inacceptables.

Cet exemple nécessite Common Crypto
Il est nécessaire d'avoir un en-tête de pontage pour le projet:
#import <CommonCrypto/CommonCrypto.h>
Ajoutez le Security.framework au projet.

Paramètres:

```
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
}
```

Exemple d'utilisation:

```
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)")
```

Exemple de sortie:

```
derivedKey (SHA1): <6b9d4fa3 0385d128 f6d196ee 3f1d6dbf>
```

Dérivation de clés par mot de passe 2 (Swift 2.3)

Voir l'exemple de Swift 3 pour les informations d'utilisation et les notes

```
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
}
```

Exemple d'utilisation:

```
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))")
```

Exemple de sortie:

```
derivedKey (SHA1):   <6b9d4fa3 0385d128 f6d196ee 3f1d6dbf>
```

Étalonnage par dérivation de clé par mot de passe (Swift 2.3)

Voir l'exemple de Swift 3 pour les informations d'utilisation et les notes

```
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
}
```

Exemple d'utilisation:

```
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)")
```

Exemple de sortie:

    Pour un délai de 100 ms, tours: 94339

Calibrage par dérivation de clé par mot de passe (Swift 3)

Déterminez le nombre de tours PRF à utiliser pour un délai spécifique sur la plate-forme

actuelle.

Plusieurs paramètres sont définis par défaut sur des valeurs représentatives qui ne devraient pas affecter matériellement le nombre de tours.

```
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
}
```

Exemple d'utilisation:

```
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)")
```

Exemple de sortie:

```
For 100 msec delay, rounds: 93457
```

Lire Dérivation de clé PBKDF2 en ligne: https://riptutorial.com/fr/swift/topic/7053/derivation-de-cle-pbkdf2

Chapitre 17: Des classes

Remarques

Le init() est une méthode spéciale dans les classes qui est utilisée pour déclarer un initialiseur pour la classe. Plus d'informations peuvent être trouvées ici: Initialiseurs

Examples

Définir une classe

Vous définissez une classe comme ceci:

```
class Dog {}
```

Une classe peut aussi être une sous-classe d'une autre classe:

```
class Animal {}
class Dog: Animal {}
```

Dans cet exemple, Animal pourrait également être un protocole auquel Dog conforme.

Sémantique de référence

Les classes sont **des types de référence** , ce qui signifie que plusieurs variables peuvent faire référence à la même instance.

```
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"
```

Comme les classes sont des types de référence, même si la classe est une constante, ses propriétés de variable peuvent toujours être modifiées.

```
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. */
```

Testez si deux objets sont *identiques* (pointez sur la même instance) en utilisant === :

```
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
```

Propriétés et méthodes

Les classes peuvent définir des propriétés que les instances de la classe peuvent utiliser. Dans cet exemple, Dog a deux propriétés: name et dogYearAge :

```
class Dog {
    var name = ""
    var dogYearAge = 0
}
```

Vous pouvez accéder aux propriétés avec une syntaxe à point:

```
let dog = Dog()
print(dog.name)
print(dog.dogYearAge)
```

Les classes peuvent également définir des méthodes pouvant être appelées sur les instances, elles sont déclarées similaires aux fonctions normales, juste à l'intérieur de la classe:

```
class Dog {
    func bark() {
        print("Ruff!")
    }
}
```

Les méthodes d'appel utilisent également la syntaxe à points:

```
dog.bark()
```

Classes et héritage multiple

Swift ne prend pas en charge l'héritage multiple. En d'autres termes, vous ne pouvez pas hériter de plus d'une classe.

```
class Animal { ... }
class Pet { ... }

class Dog: Animal, Pet { ... } // This will result in a compiler error.
```

Au lieu de cela, vous êtes encouragé à utiliser la composition lors de la création de vos types. Cela peut être accompli en utilisant des protocoles .

deinit

```
class ClassA {

    var timer: NSTimer!

    init() {
        // initialize timer
    }

    deinit {
        // code
        timer.invalidate()
    }
}
```

Lire Des classes en ligne: https://riptutorial.com/fr/swift/topic/459/des-classes

Chapitre 18: Design Patterns - Créatif

Introduction

Les modèles de conception sont des solutions générales aux problèmes fréquents dans le développement de logiciels. Les éléments suivants sont des modèles de meilleures pratiques normalisées en matière de structuration et de conception du code, ainsi que des exemples de contextes communs dans lesquels ces modèles de conception seraient appropriés.

Les motifs de conception créatifs abstraite l'instanciation des objets pour rendre un système plus indépendant du processus de création, de composition et de représentation.

Examples

Singleton

Les singletons sont un modèle de conception fréquemment utilisé qui consiste en une instance unique d'une classe partagée dans un programme.

Dans l'exemple suivant, nous créons une propriété static contenant une instance de la classe Foo . N'oubliez pas qu'une propriété static est partagée entre tous les objets d'une classe et ne peut pas être remplacée par un sous-classement.

```
public class Foo
{
    static let shared = Foo()

    // Used for preventing the class from being instantiated directly
    private init() {}

    func doSomething()
    {
        print("Do something")
    }
}
```

Usage:

```
Foo.shared.doSomething()
```

Veillez à vous souvenir de l'initialiseur private :

> Cela garantit que vos singletons sont vraiment uniques et empêche les objets extérieurs de créer leurs propres instances de votre classe grâce au contrôle d'accès. Comme tous les objets sont livrés avec un initialiseur public par défaut dans Swift, vous devez remplacer votre init et le rendre privé. KrakenDev

Méthode d'usine

> Dans la programmation basée sur les classes, le modèle de méthode de fabrique est un modèle de création qui utilise des méthodes d'usine pour résoudre le problème de la création d'objets sans avoir à spécifier la classe exacte de l'objet à créer. Référence 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)
```

En faisant cela, nous ne dépendons pas de l'implémentation réelle de la classe, ce qui rend l' sender() complètement transparent à qui le consomme.

Dans ce cas, tout ce que nous devons savoir, c'est qu'un expéditeur va gérer la livraison et expose une méthode appelée send() . Il y a plusieurs autres avantages: réduire le couplage des classes, faciliter le test, ajouter de nouveaux comportements sans avoir à changer qui en consomme.

> Dans la conception orientée objet, les interfaces fournissent des couches d'abstraction qui facilitent l'explication conceptuelle du code et créent une barrière empêchant les dépendances. Référence Wikipedia

Observateur

> Le motif de l'observateur est l'endroit où un objet, appelé le sujet, maintient une liste de ses dépendants, appelés observateurs, et les avertit automatiquement de tout changement d'état, généralement en appelant l'une de leurs méthodes. Il est principalement utilisé pour mettre en œuvre des systèmes de gestion d'événements distribués. Le modèle Observer est également un élément clé du modèle architectural familier modèle-vue-contrôleur (MVC). Référence Wikipedia

Fondamentalement, le modèle d'observateur est utilisé lorsque vous avez un objet qui peut avertir les observateurs de certains comportements ou changements d'état.

Tout d'abord, créons une référence globale (en dehors d'une classe) pour le Centre de notifications:

```
let notifCentre = NotificationCenter.default
```

Super maintenant, nous pouvons appeler cela de n'importe où. Nous voudrions alors enregistrer une classe en tant qu'observateur ...

```
notifCentre.addObserver(self, selector: #selector(self.myFunc), name: "myNotification",
object: nil)
```

Cela ajoute la classe en tant qu'observateur pour "readForMyFunc". Il indique également que la fonction myFunc doit être appelée lorsque cette notification est reçue. Cette fonction doit être écrite dans la même classe:

```
func myFunc(){
    print("The notification has been received")
}
```

> L'un des avantages de ce modèle est que vous pouvez ajouter de nombreuses classes en tant qu'observateurs et effectuer ainsi de nombreuses actions après une notification.

La notification peut maintenant être simplement envoyée (ou affichée si vous préférez) depuis presque n'importe où dans le code avec la ligne:

```
notifCentre.post(name: "myNotification", object: nil)
```

> Vous pouvez également transmettre des informations avec la notification en tant que dictionnaire

```
let myInfo = "pass this on"
notifCentre.post(name: "myNotification", object: ["moreInfo":myInfo])
```

Mais alors vous devez ajouter une notification à votre fonction:

```
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
}
```

Chaîne de responsabilité

> Dans la conception orientée objet, le modèle de chaîne de responsabilité est un modèle de conception constitué d'une source d'objets de command et d'une série d'objets de processing . Chaque objet de processing contient une logique qui définit les types d'objets de commande qu'il peut gérer; les autres sont transmis au prochain objet de processing de la chaîne. Un mécanisme existe également pour ajouter de nouveaux objets de processing à la fin de cette chaîne. Wikipédia

Mise en place des cours constituant la chaîne de responsabilité.

Nous créons d'abord une interface pour tous les objets 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)
    }
  }
}
```

Ensuite, nous créons l'objet de command .

```
struct PurchaseRequest {
  var amount : Float
  var purpose : String
}
```

Enfin, créer des objets constituant la chaîne de responsabilité.

```
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?
}
```

Initier et enchaîner ensemble:

```
let manager = ManagerPower()
let director = DirectorPower()
let president = PresidentPower()

manager.successor = director
director.successor = president
```

Le mécanisme pour chaîner des objets ici est l'accès à la propriété

Création d'une requête pour l'exécuter:

```
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
```

Itérateur

> En programmation informatique, un itérateur est un objet qui permet à un programmeur de traverser un conteneur, en particulier des listes. Wikipédia

```
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)
  }
}
```

Et exemple d'utilisation serait

```
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)")
}
```

Motif de constructeur

> Le modèle de générateur est un **modèle de conception de logiciel de création d'objet** . Contrairement au modèle de fabrique abstrait et au modèle de méthode de fabrique dont l’intention est de permettre le polymorphisme, l’intention du modèle de générateur est de trouver une solution à l’anti-pattern du constructeur télescopique. L'anti-modèle du constructeur télescopique se produit lorsque l'augmentation de la combinaison de paramètres du constructeur d'objet mène à une liste exponentielle de constructeurs. Au lieu d'utiliser de nombreux constructeurs, le modèle de générateur utilise un autre objet, un générateur, qui reçoit chaque paramètre d'initialisation étape par étape et renvoie ensuite l'objet construit résultant.
>
> *-Wikipédia*

L'objectif principal du modèle de générateur est de configurer une configuration par défaut pour un objet depuis sa création. C'est un intermédiaire entre l'objet à construire et tous les autres objets liés à sa construction.

Exemple:

Pour le rendre plus clair, examinons un exemple de *constructeur de voiture* .

Considérez que nous avons une classe *Car* contient de nombreuses options pour créer un objet, telles que:

- Couleur.
- Nombre de places.
- Nombre de roues
- Type.
- Type d'engin
- Moteur.
- Disponibilité des airbags.

```
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

    }
}
```

Créer un objet de voiture:

```
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
*/
```

Le **problème** se pose lors de la création d'un objet de voiture: la voiture nécessite la création de nombreuses données de configuration.

Pour appliquer le modèle de générateur, les paramètres d'initialisation doivent avoir des valeurs par défaut *modifiables si nécessaire* .

Classe 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 classe CarBuilder définit les propriétés pouvant être modifiées pour modifier les valeurs de l'objet de voiture créé.

Construisons de nouvelles voitures en utilisant le 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
*/
```

L' **avantage** de l'application du modèle de générateur est la facilité de création d'objets qui doivent contenir beaucoup de configurations en définissant des valeurs par défaut, ainsi que la facilité de modification de ces valeurs par défaut.

Continuer:

Comme bonne pratique, toutes les propriétés nécessitant des valeurs par défaut doivent être dans un *protocole séparé* , qui doit être implémenté par la classe elle-même et son générateur.

En s'appuyant sur notre exemple, créons un nouveau protocole appelé 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)
    }
}
```

L'avantage de déclarer les propriétés nécessitant une valeur par défaut dans un protocole est de forcer l'implémentation de toute nouvelle propriété ajoutée. Lorsqu'une classe est conforme à un protocole, elle doit déclarer toutes ses propriétés / méthodes.

Considérez qu'il y a une nouvelle fonctionnalité requise qui devrait être ajoutée au modèle de création d'une voiture appelée "nom de la batterie":

```
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 }
}
```

Après avoir ajouté la nouvelle propriété, notez que deux erreurs de compilation se produiront, notifiant que la conformité au protocole CarBluePrint nécessite de déclarer la propriété 'batteryName'. Cela garantit que CarBuilder déclarera et définira une valeur par défaut pour la propriété batteryName .

Après avoir ajouté batteryName nouvelle propriété à CarBluePrint protocole, la mise en œuvre des deux Car et CarBuilder classes devraient être:

```
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)
    }
}
```

Encore une fois, construisons de nouvelles voitures en utilisant le 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
 */
```

Lire Design Patterns - Créatif en ligne: https://riptutorial.com/fr/swift/topic/4941/design-patterns---creatif

Chapitre 19: Design Patterns - Structural

Introduction

Les modèles de conception sont des solutions générales aux problèmes fréquents dans le développement de logiciels. Les éléments suivants sont des modèles de meilleures pratiques normalisées en matière de structuration et de conception du code, ainsi que des exemples de contextes communs dans lesquels ces modèles de conception seraient appropriés.

Les modèles de conception structurelle se concentrent sur la composition des classes et des objets pour créer des interfaces et obtenir de meilleures fonctionnalités.

Examples

Adaptateur

Les adaptateurs permettent de convertir l'interface d'une classe donnée, appelée **Adaptee** , en une autre interface, appelée **Target** . Les opérations sur la cible sont appelées par un **client** et ces opérations sont *adaptées* par l'adaptateur et transmises à l'adaptee.

Dans Swift, les adaptateurs peuvent souvent être formés à l'aide de protocoles. Dans l'exemple suivant, un client pouvant communiquer avec la cible a la possibilité d'exécuter des fonctions de la classe Adaptee en utilisant un adaptateur.

```
// 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()
    }
}
```

Exemple de flux d'un adaptateur unidirectionnel: Client -> Target -> Adapter -> Adaptee

Les adaptateurs peuvent également être bidirectionnels, appelés **adaptateurs bidirectionnels** . Un adaptateur bidirectionnel peut être utile lorsque deux clients différents doivent afficher un objet différemment.

Façade

Une **façade** fournit une interface unifiée et de haut niveau aux interfaces de sous-système. Cela permet un accès plus simple et plus sûr aux installations plus générales d'un sous-système.

Voici un exemple de façade utilisé pour définir et récupérer des objets dans 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!
    }

}
```

L'utilisation pourrait ressembler à ceci:

```
Defaults.set("Beyond all recognition.", forKey:"fooBar")
Defaults.object(forKey: "fooBar")
```

Les complexités liées à l'accès à l'instance partagée et à la synchronisation des UserDefaults sont masquées pour le client, et cette interface est accessible depuis n'importe où dans le programme.

Lire Design Patterns - Structural en ligne: https://riptutorial.com/fr/swift/topic/9497/design-patterns---structural

Chapitre 20: Dictionnaires

Remarques

Certains exemples de cette rubrique peuvent avoir un ordre différent lorsqu'ils sont utilisés, car l'ordre du dictionnaire n'est pas garanti.

Examples

Déclaration de dictionnaires

Les dictionnaires sont une collection non ordonnée de clés et de valeurs. Les valeurs se rapportent à des clés uniques et doivent être du même type.

Lors de l'initialisation d'un dictionnaire, la syntaxe complète est la suivante:

```
var books : Dictionary<Int, String> = Dictionary<Int, String>()
```

Bien que ce soit un moyen plus concis d'initialiser:

```
var books = [Int: String]()
// or
var books: [Int: String] = [:]
```

Déclarez un dictionnaire avec des clés et des valeurs en les spécifiant dans une liste séparée par des virgules. Les types peuvent être déduits des types de clés et de valeurs.

```
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"]
```

Modification des dictionnaires

Ajouter une clé et une valeur à un dictionnaire

```
var books = [Int: String]()
//books = [:]
books[5] = "Book 5"
//books = [5: "Book 5"]
books.updateValue("Book 6", forKey: 5)
//[5: "Book 6"]
```

updateValue renvoie la valeur d'origine s'il en existe une ou nulle.

```
let previousValue = books.updateValue("Book 7", forKey: 5)
//books = [5: "Book 7"]
//previousValue = "Book 6"
```

Supprimer la valeur et leurs clés avec une syntaxe similaire

```
books[5] = nil
//books [:]
books[6] = "Deleting from Dictionaries"
```

```
//books = [6: "Deleting from Dictionaries"]
let removedBook = books.removeValueForKey(6)
//books = [:]
//removedValue = "Deleting from Dictionaries"
```

Accès aux valeurs

Une valeur dans un Dictionary est accessible à l'aide de sa clé:

```
var books: [Int: String] = [1: "Book 1", 2: "Book 2"]
let bookName = books[1]
//bookName = "Book 1"
```

Les valeurs d'un dictionnaire peuvent être itérées à l'aide de la propriété values :

```
for book in books.values {
    print("Book Title: \(book)")
}
//output: Book Title: Book 2
//output: Book Title: Book 1
```

De même, les clés d'un dictionnaire peuvent être itérées en utilisant sa propriété keys :

```
for bookNumbers in books.keys {
    print("Book number: \(bookNumber)")
}
// outputs:
// Book number: 1
// Book number: 2
```

Pour obtenir toutes key paires de key et de value correspondant les unes aux autres (vous n'obtiendrez pas le bon ordre car il s'agit d'un dictionnaire)

```
for (book,bookNumbers)in books{
print("\(book)  \(bookNumbers)")
}
// outputs:
// 2  Book 2
// 1  Book 1
```

Notez qu'un Dictionary , contrairement à un Array , par nature non ordonné, c'est-à-dire qu'il n'y a aucune garantie sur la commande pendant l'itération.

Si vous souhaitez accéder à plusieurs niveaux d'un dictionnaire, utilisez une syntaxe d'indice répétée.

```
// 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"
```

Modifier la valeur du dictionnaire à l'aide de la clé

```
var dict = ["name": "John", "surname": "Doe"]
// Set the element with key: 'name' to 'Jane'
dict["name"] = "Jane"
print(dict)
```

Obtenir toutes les clés dans le dictionnaire

```
let myAllKeys = ["name" : "Kirit" , "surname" : "Modi"]
let allKeys = Array(myAllKeys.keys)
print(allKeys)
```

Fusionner deux dictionnaires

```
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
    }
}
```

Lire Dictionnaires en ligne: https://riptutorial.com/fr/swift/topic/310/dictionnaires

Chapitre 21: Enregistrement dans Swift

Remarques

println et debugPrintln supprimés dans Swift 2.0.

Sources:

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

Debug Print

Debug Print affiche la représentation d'instance la plus appropriée pour le débogage.

```
print("Hello")
debugPrint("Hello")

let dict = ["foo": 1, "bar": 2]

print(dict)
debugPrint(dict)
```

Les rendements

```
>>> Hello
>>> "Hello"
>>> [foo: 1, bar: 2]
>>> ["foo": 1, "bar": 2]
```

Ces informations supplémentaires peuvent être très importantes, par exemple:

```
let wordArray = ["foo", "bar", "food, bars"]

print(wordArray)
debugPrint(wordArray)
```

Les rendements

```
>>> [foo, bar, food, bars]
>>> ["foo", "bar", "food, bars"]
```

Notez que dans la première sortie, il semble qu'il y ait 4 éléments dans le tableau, contre 3. Pour des raisons comme celles-ci, il est préférable, lors du débogage, d'utiliser debugPrint

Mise à jour des valeurs de débogage et d'impression des classes

```
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
```

déverser

dump imprime le contenu d'un objet par réflexion (mise en miroir).

Vue détaillée d'un tableau:

```
let names = ["Joe", "Jane", "Jim", "Joyce"]
dump(names)
```

Impressions:

```
▿ 4 éléments
- [0]: Joe
- [1]: Jane
- [2]: Jim
- [3]: Joyce
```

Pour un dictionnaire:

```
let attributes = ["foo": 10, "bar": 33, "baz": 42]
dump(attributes)
```

Impressions:

```
Pairs 3 paires clé / valeur
▿ [0]: (2 éléments)
- .0: bar
- 0,1: 33
▿ [1]: (2 éléments)
- .0: baz
- .1: 42
▿ [2]: (2 éléments)
- .0: foo
- .1: 10
```

dump est déclaré comme dump(_:name:indent:maxDepth:maxItems:) .

Le premier paramètre n'a pas d'étiquette.

Il y a d'autres paramètres disponibles, comme name pour définir une étiquette pour l'objet inspecté:

```
dump(attributes, name: "mirroring")
```

Impressions:

```
▿ miroir: 3 paires clé / valeur
▿ [0]: (2 éléments)
```

```
    - .0: bar
    - 0,1: 33
    ▿ [1]: (2 éléments)
    - .0: baz
    - .1: 42
    ▿ [2]: (2 éléments)
    - .0: foo
    - .1: 10
```

Vous pouvez également choisir d'imprimer uniquement un certain nombre d'éléments avec maxItems: pour analyser l'objet jusqu'à une certaine profondeur avec maxDepth: et pour modifier l'indentation des objets imprimés avec un indent: maxDepth:

print () vs dump ()

Beaucoup d'entre nous commencent à déboguer avec print() . Disons que nous avons une telle classe:

```
class Abc {
    let a = "aa"
    let b = "bb"
}
```

et nous avons une instance de Abc comme ça:

```
let abc = Abc()
```

Lorsque nous exécutons print() sur la variable, la sortie est

```
App.Abc
```

pendant que dump() sort

```
App.Abc #0
- a: "aa"
- b: "bb"
```

Comme on l'a vu, dump() affiche toute la hiérarchie de classe, tandis que print() génère simplement le nom de la classe.

Par conséquent, dump() est particulièrement utile pour le débogage de l'interface utilisateur.

```
let view = UIView(frame: CGRect(x: 0, y: 0, width: 100, height: 100))
```

Avec dump(view) nous obtenons:

```
 - <UIView: 0x108a0cde0; frame = (0 0; 100 100); layer = <CALayer: 0x159340cb0>> #0
    - super: UIResponder
      - NSObject
```

Pendant l' print(view) nous obtenons:

```
<UIView: 0x108a0cde0; frame = (0 0; 100 100); layer = <CALayer: 0x159340cb0>>
```

Il y a plus d'informations sur la classe avec dump() , et c'est donc plus utile pour déboguer la classe elle-même.

imprimer vs NSLog

En swift, nous pouvons utiliser les fonctions print() et NSLog() pour imprimer quelque chose sur la console Xcode.

Mais il y a beaucoup de différences dans les fonctions print() et NSLog() , telles que:

1 TimeStamp: NSLog() imprimera l'horodatage avec la chaîne que nous lui avons transmise, mais print() n'imprimera pas d'horodatage.
par exemple

```
let array = [1, 2, 3, 4, 5]
print(array)
NSLog(array.description)
```

Sortie:

```
[1, 2, 3, 4, 5]
2017-05-31 13: 14: 38.582 ProjetName [2286: 7473287] [1, 2, 3, 4, 5]
```

Il va également imprimer **ProjectName** avec l'horodatage.

2 Uniquement en chaîne: NSLog() prend uniquement la chaîne en tant qu'entrée, mais print() peut imprimer tout type d'entrée qui lui est transmis.
par exemple

```
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 Performance: la fonction NSLog() est très **lente** comparée à la fonction print() .

4 Synchronisation: NSLog() gère l'utilisation simultanée de l'environnement multi-threading et imprime la sortie sans la chevaucher. Mais print() ne traitera pas de tels cas et de tels problèmes lors de la diffusion des résultats.

5 Device Console: les sorties NSLog() sur la console du périphérique également, nous pouvons voir cette sortie en connectant notre appareil à Xcode. print() n'imprimera pas la sortie sur la console du périphérique.

Lire Enregistrement dans Swift en ligne:
https://riptutorial.com/fr/swift/topic/3966/enregistrement-dans-swift

Chapitre 22: Ensembles

Examples

Déclarations

Les ensembles sont des collections non ordonnées de valeurs uniques. Les valeurs uniques doivent être du même type.

```
var colors = Set<String>()
```

Vous pouvez déclarer un ensemble avec des valeurs en utilisant la syntaxe littérale du tableau.

```
var favoriteColors: Set<String> = ["Red", "Blue", "Green", "Blue"]
// {"Blue", "Green", "Red"}
```

Modification de valeurs dans un ensemble

```
var favoriteColors: Set = ["Red", "Blue", "Green"]
//favoriteColors = {"Blue", "Green", "Red"}
```

Vous pouvez utiliser la méthode insert(_:) pour ajouter un nouvel élément dans un ensemble.

```
favoriteColors.insert("Orange")
//favoriteColors = {"Red", "Green", "Orange", "Blue"}
```

Vous pouvez utiliser la méthode remove(_:) pour supprimer un élément d'un ensemble. Il renvoie une valeur contenant facultative qui a été supprimée ou nulle si la valeur ne figurait pas dans l'ensemble.

```
let removedColor = favoriteColors.remove("Red")
//favoriteColors = {"Green", "Orange", "Blue"}
// removedColor = Optional("Red")

let anotherRemovedColor = favoriteColors.remove("Black")
// anotherRemovedColor = nil
```

Vérifier si un ensemble contient une valeur

```
var favoriteColors: Set = ["Red", "Blue", "Green"]
//favoriteColors = {"Blue", "Green", "Red"}
```

Vous pouvez utiliser la méthode contains(_:) pour vérifier si un ensemble contient une valeur. Il retournera true si l'ensemble contient cette valeur.

```
if favoriteColors.contains("Blue") {
    print("Who doesn't like blue!")
}
// Prints "Who doesn't like blue!"
```

Effectuer des opérations sur des ensembles

Valeurs communes aux deux ensembles:

Vous pouvez utiliser la méthode intersect(_:) pour créer un nouvel ensemble contenant toutes les valeurs communes aux deux ensembles.

```
let favoriteColors: Set = ["Red", "Blue", "Green"]
let newColors: Set = ["Purple", "Orange", "Green"]

let intersect = favoriteColors.intersect(newColors) // a AND b
// intersect = {"Green"}
```

Toutes les valeurs de chaque ensemble:

Vous pouvez utiliser la méthode union(_:) pour créer un nouvel ensemble contenant toutes les valeurs uniques de chaque ensemble.

```
let union = favoriteColors.union(newColors) // a OR b
// union = {"Red", "Purple", "Green", "Orange", "Blue"}
```

Notez que la valeur "Vert" n'apparaît qu'une seule fois dans le nouvel ensemble.

Des valeurs qui n'existent pas dans les deux ensembles:

Vous pouvez utiliser la méthode exclusiveOr(_:) pour créer un nouvel ensemble contenant les valeurs uniques des deux ensembles, mais pas des deux.

```
let exclusiveOr = favoriteColors.exclusiveOr(newColors) // a XOR b
// exclusiveOr = {"Red", "Purple", "Orange", "Blue"}
```

Remarquez que la valeur "Green" n'apparaît pas dans le nouvel ensemble, car elle figurait dans les deux ensembles.

Valeurs qui ne sont pas dans un ensemble:

Vous pouvez utiliser la méthode de subtract(_:) pour créer un nouvel ensemble contenant des valeurs qui ne figurent pas dans un ensemble spécifique.

```
let subtract = favoriteColors.subtract(newColors) // a - (a AND b)
// subtract = {"Blue", "Red"}
```

Remarquez que la valeur "Green" n'apparaît pas dans le nouvel ensemble, car elle figurait également dans le deuxième ensemble.

Ajout de valeurs de mon propre type à un ensemble

Afin de définir un Set de votre propre type, vous devez Hashable votre type à Hashable

```
struct Starship: Hashable {
    let name: String
    var hashValue: Int { return name.hashValue }
}

func ==(left:Starship, right: Starship) -> Bool {
    return left.name == right.name
}
```

Maintenant, vous pouvez créer un Set de Starship(s)

```
let ships : Set<Starship> = [Starship(name:"Enterprise D"), Starship(name:"Voyager"),
Starship(name:"Defiant") ]
```

CountedSet

3.0

Swift 3 introduit la classe CountedSet (c'est la version Swift de la NSCountedSet Objective-C).

CountedSet, tel que suggéré par le nom, conserve le nombre de fois qu'une valeur est présente.

```
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
```

Lire Ensembles en ligne: https://riptutorial.com/fr/swift/topic/371/ensembles

Chapitre 23: Enums

Remarques

Tout comme les classes et les classes, les énumérations sont des types de valeurs et sont copiées au lieu d'être référencées lorsqu'elles sont transmises.

Pour plus d'informations sur les énumérations, voir Le langage de programmation Swift .

Examples

Énumérations de base

Une énumération fournit un ensemble de valeurs connexes:

```
enum Direction {
    case up
    case down
    case left
    case right
}

enum Direction { case up, down, left, right }
```

Les valeurs Enum peuvent être utilisées par leur nom complet, mais vous pouvez omettre le nom du type lorsqu'il peut être déduit:

```
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 méthode la plus fondamentale pour comparer / extraire des valeurs enum consiste à utiliser une instruction 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
}
```

Les Hashable simples sont automatiquement Hashable , Equatable et ont des conversions de chaînes:

```
if dir == .down { ... }
```

```
let dirs: Set<Direction> = [.right, .left]

print(Direction.up)  // prints "up"
debugPrint(Direction.up)  // prints "Direction.up"
```

Enums avec des valeurs associées

Les cas d'énumération peuvent contenir une ou plusieurs **charges utiles** (**valeurs associées**):

```
enum Action {
    case jump
    case kick
    case move(distance: Float)  // The "move" case has an associated distance
}
```

Le payload doit être fourni lors de l'instanciation de la valeur enum:

```
performAction(.jump)
performAction(.kick)
performAction(.move(distance: 3.3))
performAction(.move(distance: 0.5))
```

L'instruction switch peut extraire la valeur associée:

```
switch action {
case .jump:
    ...
case .kick:
    ...
case .move(let distance):  // or case let .move(distance):
    print("Moving: \(distance)")
}
```

Une extraction de cas unique peut être effectuée en utilisant if case :

```
if case .move(let distance) = action {
    print("Moving: \(distance)")
}
```

La syntaxe du guard case peut être utilisée pour l'extraction ultérieure:

```
guard case .move(let distance) = action else {
    print("Action is not move")
    return
}
```

Les énumérations associées à des valeurs ne sont pas Equatable par défaut. L'implémentation de l'opérateur == doit être effectuée manuellement:

```
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
}
```

Charges indirectes

Normalement, les énumérations ne peuvent pas être récursives (car elles nécessiteraient un stockage infini):

```
enum Tree<T> {
    case leaf(T)
    case branch(Tree<T>, Tree<T>)  // error: recursive enum 'Tree<T>' is not marked 'indirect'
}
```

Le mot-clé **indirect** fait que l'enum stocke ses données utiles avec une couche d'indirection plutôt que de les stocker en ligne. Vous pouvez utiliser ce mot clé sur un seul cas:

```
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 fonctionne également sur l'ensemble de l'énumération, rendant chaque cas indirect lorsque nécessaire:

```
indirect enum Tree<T> {
    case leaf(T)
    case branch(Tree<T>, Tree<T>)
}
```

Valeurs brutes et hachages

Les énumérations sans données utiles peuvent avoir *des valeurs brutes* de tout type littéral:

```
enum Rotation: Int {
    case up = 0
    case left = 90
    case upsideDown = 180
    case right = 270
}
```

Les énumérations sans type spécifique n'exposent pas la propriété rawValue

```
enum Rotation {
    case up
    case right
    case down
    case left
}

let foo = Rotation.up
foo.rawValue //error
```

Les valeurs brutes entières sont supposées commencer à 0 et augmenter de manière monotone:

```
enum MetasyntacticVariable: Int {
    case foo  // rawValue is automatically 0
    case bar  // rawValue is automatically 1
    case baz = 7
    case quux  // rawValue is automatically 8
}
```

Les valeurs brutes peuvent être synthétisées automatiquement:

```
enum MarsMoon: String {
    case phobos  // rawValue is automatically "phobos"
    case deimos  // rawValue is automatically "deimos"
}
```

Un enum à valeur brute est automatiquement conforme à RawRepresentable . Vous pouvez obtenir la valeur brute correspondante avec .rawValue :

```
func rotate(rotation: Rotation) {
    let degrees = rotation.rawValue
    ...
}
```

Vous pouvez également créer une énumération à *partir d'* une valeur brute à l'aide d' 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 vous souhaitez obtenir la valeur de hachage d'une enum spécifique, vous pouvez accéder à sa hashValue. La valeur de hachage renverra l'index de l'enum à partir de zéro.

```
let quux = MetasyntacticVariable(rawValue: 8)// rawValue is 8
quux?.hashValue //hashValue is 3
```

Initialiseurs

Enums peuvent avoir des méthodes d'initialisation personnalisées qui peuvent être plus utiles que l' init?(rawValue:) par défaut init?(rawValue:) . Enums peut également stocker des valeurs. Cela peut être utile pour stocker les valeurs avec lesquelles ils ont été initialisés et pour récupérer cette valeur ultérieurement.

```
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
        }
    }
}
```

En utilisant cet initialiseur, nous pouvons faire ceci:

```
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
```

Les énumérations partagent de nombreuses fonctionnalités avec les classes et les structures

Enums Swift sont beaucoup plus puissants que certains de leurs homologues dans d'autres langues, telles que C. Ils partagent de nombreuses fonctionnalités avec des classes et des structures , telles que la définition d' initialiseurs , de propriétés calculées , de méthodes d'instance , de conformités de protocole et d' extensions .

```
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
```

Enumérations imbriquées

Vous pouvez imbriquer les énumérations les unes dans les autres, cela vous permet de structurer les énumérations hiérarchiques pour qu'elles soient plus organisées et claires.

```
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
    }
}
```

Et vous pouvez l'utiliser comme ça:

```
let instrment1 = Orchestra.Strings.viola
let instrment2 = Orchestra.Keyboards.piano
```

Lire Enums en ligne: https://riptutorial.com/fr/swift/topic/224/enums

Chapitre 24: Fermetures

Syntaxe

- var closureVar: (<paramètres>) -> (<returnType>) // En tant que variable ou type de propriété
- typealias ClosureType = (<paramètres>) -> (<returnType>)
- {[<captureList>] (<paramètres>) <throws-ness> -> <returnType> dans <instructions>} // Syntaxe de fermeture complète

Remarques

Pour plus d'informations sur les fermetures Swift, consultez la documentation Apple .

Examples

Bases de fermeture

Les fermetures (également appelées **blocs** ou **lambdas**) sont des morceaux de code qui peuvent être stockés et transmis dans votre programme.

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

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

Comme les autres fonctions, les fermetures peuvent accepter des arguments et des résultats de retour ou des erreurs de lancement:

```
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"
```

Les fermetures peuvent **capturer des** valeurs de leur portée:

```
// 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
```

Les fermetures peuvent être passées directement dans les fonctions:

```
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()
```

Variations de syntaxe

La syntaxe de fermeture de base est

> { [*liste de capture*] (*paramètres*) *throws-ness* -> *type de retour* in *corps* } .

Beaucoup de ces parties peuvent être omises, il existe donc plusieurs manières équivalentes d'écrire des fermetures 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 liste de capture peut être omise si elle est vide.
- Les paramètres n'ont pas besoin d'annotations de type si leurs types peuvent être déduits.
- Le type de retour n'a pas besoin d'être spécifié s'il peut être déduit.
- Les paramètres ne doivent pas être nommés; au lieu de cela, ils peuvent être référés avec $0 $1 $2 , etc.
- Si la fermeture contient une seule expression, dont la valeur doit être renvoyée, le mot-clé de return peut être omis.
- Si la fermeture est supposée générer une erreur, est écrite dans un contexte qui attend une fermeture de lancement ou ne throws pas d'erreur, les throws peuvent être omis.

```
// 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 }
```

Passer des fermetures dans des fonctions

Les fonctions peuvent accepter des fermetures (ou d'autres fonctions) en tant que paramètres:

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

Syntaxe de clôture

Si le dernier paramètre d'une fonction est une fermeture, les accolades { / } peuvent être écrites **après** l'appel de la fonction:

```
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 le seul argument d'une fonction est une fermeture, vous pouvez également omettre la paire de parenthèses () lorsque vous l'appelez avec la syntaxe de clôture:

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

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

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

Paramètres @noescape

Les paramètres de fermeture marqués @noescape sont garantis pour s'exécuter avant le retour de l'appel de la fonction, donc en utilisant self. n'est pas nécessaire à l'intérieur du corps de fermeture:

```
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 }
    }
}
```

Swift 3 note:

Notez que dans Swift 3, vous ne marquez plus les blocs comme @noescape. Les blocs **ne** s'échappent plus par défaut. Dans Swift 3, au lieu de marquer une fermeture comme ne s'échappant pas, vous marquez un paramètre de fonction qui est une clôture d'échappement en évitant d'utiliser le mot clé "@escaping".

---

throws et rethrows

Les fermetures, comme les autres fonctions, peuvent générer des erreurs :

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

La fonction peut bien entendu transmettre l'erreur à son correspondant:

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

Cependant, si le bloc transmis *ne* lance *pas* , l'appelant est toujours bloqué par une fonction de lancement:

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

La solution est **rethrows** , qui **rethrows** que la fonction ne peut lancer que **si son paramètre de fermeture lance** :

```
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 }
```

De nombreuses fonctions de bibliothèque standard utilisent des rethrows , notamment map() , filter() et indexOf() .

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

Lorsqu'une fermeture capture un type de référence (une instance de classe), elle contient une référence forte par défaut:

```
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 **liste de capture de** la fermeture peut être utilisée pour spécifier une référence faible ou non:

```
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.
```

Pour plus d'informations, reportez-vous à la rubrique Gestion de la mémoire et à la section Comptage automatique des références du langage de programmation Swift.

Conserver les cycles

Si un objet retient une fermeture, qui contient également une référence forte à l'objet, il s'agit d'un **cycle de conservation** . À moins que le cycle ne soit rompu, la mémoire stockant l'objet et la fermeture sera perdue (jamais récupérée).

```
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)")
        }
    }
}
```

Utilisation de fermetures pour le codage asynchrone

Les fermetures sont souvent utilisées pour des tâches asynchrones, par exemple lors de l'extraction de données à partir d'un site 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()
}
```

Cette fonction est asynchrone, elle ne bloquera donc pas le thread sur lequel elle est appelée (elle ne gèlera pas l'interface si elle est appelée sur le thread principal de votre application 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")
```

Comme la tâche est asynchrone, la sortie ressemblera généralement à ceci:

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

Étant donné que le code à l'intérieur de la fermeture, `print("2. Fetched data")` , ne sera pas appelé tant que les données de l'URL ne sont pas extraites.

Fermetures et alias de type

Une fermeture peut être définie avec une `typealias` . Cela fournit un espace réservé de type pratique si la même signature de fermeture est utilisée à plusieurs endroits. Par exemple, les rappels de requêtes réseau courants ou les gestionnaires d'événements d'interface utilisateur sont d'excellents candidats pour être "nommés" avec un alias de type.

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

Vous pouvez ensuite définir une fonction en utilisant les typealias:

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

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

Lire Fermetures en ligne: https://riptutorial.com/fr/swift/topic/262/fermetures

Chapitre 25: Fonctionne comme des citoyens de première classe à Swift

Introduction

Fonctionne en tant que membre de première classe signifie qu'il peut bénéficier de privilèges comme les objets. Il peut être affecté à une variable, transmis à une fonction en tant que paramètre ou peut être utilisé comme type de retour.

Examples

Affectation d'une fonction à une 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))
```

Sortie:

```
[3, 5, 7, 9, 11, 13, 10, 8, 6, 4, 102]
```

De même, ce qui précède pourrait être réalisé en utilisant une `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))
```

Passer la fonction comme argument à une autre fonction, créant ainsi une fonction d'ordre supérieur

```
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))
```

Sortie:

```
[3, 5, 7, 9, 11, 13, 10, 8, 6, 4, 102]
```

De même, ce qui précède pourrait être réalisé en utilisant une `closure`

```
// passing the closure directly to the function as param
print(math.performOperation(inputArray: arrayToBeProcessed, operation: { $0 * 2 }))
```

Fonction comme type de retour d'une autre fonction

```
// function as return type
print(math.performComplexOperation(valueOne: 4)(5))
```

Sortie:

9

Lire Fonctionne comme des citoyens de première classe à Swift en ligne: https://riptutorial.com/fr/swift/topic/8618/fonctionne-comme-des-citoyens-de-premiere-classe-a-swift

Chapitre 26: Fonctions Swift Advance

Introduction

Fonctions Advance comme map , flatMap , filter et reduce sont utilisés pour faire fonctionner sur différents types de collections comme matrice et le dictionnaire. Les fonctions avancées nécessitent généralement peu de code et peuvent être enchaînées afin de construire une logique complexe de manière concise.

Examples

Introduction aux fonctions avancées

Prenons un scénario pour mieux comprendre la fonction avancée,

```
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]
```

Fonction de la carte:

Utilisez map pour effectuer une boucle sur une collection et appliquer la même opération à chaque élément de la collection. La fonction map renvoie un tableau contenant les résultats de l'application d'une fonction de mappage ou de transformation à chaque élément.

```
//Fetch all the user's name from array
let arrUserName = arrUser.map({ $0.name }) // ["John", "Chan", "Morgan", "Rachel", "Katie",
"David", "Bob"]
```

Fonction Flat-Map:

L'utilisation la plus simple est, comme son nom l'indique, d'aplatir une collection de collections.

```
// Fetch all user country name & ignore nil value.
let arrCountry = arrUser.flatMap({ $0.country }) // ["USA", "UK", "USA", "USA"]
```

Fonction de filtre:

Utilisez le filtre pour parcourir une collection et renvoyer un tableau contenant uniquement les éléments correspondant à une condition d'inclusion.

```
// 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"]
```

Réduire:

Utilisez réduire pour combiner tous les éléments d'une collection afin de créer une nouvelle valeur unique.

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
```

Aplatir un tableau multidimensionnel

Pour aplatir un tableau multidimensionnel en une seule dimension, les fonctions avancées de flatMap sont utilisées. Un autre cas d'utilisation est de négliger la valeur nulle des valeurs de tableau et de mappage. Vérifions avec exemple: -

Supposons que nous ayons un tableau multidimensionnel de villes et que nous souhaitons classer la liste des noms de ville par ordre croissant. Dans ce cas, nous pouvons utiliser la fonction flatMap comme: -

```
let arrStateName = [["Alaska", "Iowa", "Missouri", "New Mexico"], ["New York", "Texas",
"Washington", "Maryland"], ["New Jersey", "Virginia", "Florida", "Colorado"]]
```

Préparer une liste à une seule dimension à partir d'un tableau multidimensionnel,

```
let arrFlatStateList = arrStateName.flatMap({ $0 }) // ["Alaska", "Iowa", "Missouri", "New
Mexico", "New York", "Texas", "Washington", "Maryland", "New Jersey", "Virginia", "Florida",
"Colorado"]
```

Pour trier les valeurs de tableau, nous pouvons utiliser une opération de chaînage ou trier un tableau à plat. Voici l'exemple ci-dessous montrant l'opération de chaînage,

```
// 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"]
```

Lire Fonctions Swift Advance en ligne: https://riptutorial.com/fr/swift/topic/9279/fonctions-swift-advance

Chapitre 27: Générer UIImage d'Initiales à partir de String

Introduction

Ceci est une classe qui générera une image UII des initiales d'une personne. Harry Potter générerait une image de HP.

Examples

InitialsImageFactory

```
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
}

}
```

Lire Générer UIImage d'Initiales à partir de String en ligne: https://riptutorial.com/fr/swift/topic/10915/generer-uiimage-d-initiales-a-partir-de-string

Chapitre 28: Génériques

Remarques

> Le code générique vous permet d'écrire des fonctions et des types flexibles et réutilisables pouvant fonctionner avec n'importe quel type, sous réserve des exigences que vous définissez. Vous pouvez écrire un code qui évite la duplication et exprime son intention de manière claire et abstraite.
>
> Les génériques sont l'une des fonctionnalités les plus puissantes de Swift, et une grande partie de la bibliothèque standard Swift est construite avec un code générique. Par exemple, les types Array et Dictionary de Swift sont tous deux des collections génériques. Vous pouvez créer un tableau Int valeurs Int ou un tableau String valeurs String , ou un tableau pour tout autre type pouvant être créé dans Swift. De même, vous pouvez créer un dictionnaire pour stocker les valeurs de tout type spécifié, et il n'y a pas de limitation quant à ce type.
>
> **Source:** Langage de programmation rapide d'Apple

Examples

Restriction des types d'espaces génériques

Il est possible de forcer les paramètres de type d'une classe générique à implémenter un protocole , par exemple, 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
    }
}
```

Chaque fois que nous créons une nouvelle MyGenericClass , le paramètre type doit implémenter le protocole Equatable (en veillant à ce que le paramètre type puisse être comparé à une autre variable du même type utilisant ==)

```
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"))
```

Les bases des génériques

Les génériques sont des espaces réservés aux types, ce qui vous permet d'écrire un code flexible pouvant être appliqué à plusieurs types. L'avantage de l'utilisation de génériques sur Any est qu'ils permettent toujours au compilateur d'appliquer une sécurité de type forte.

Un espace réservé générique est défini entre crochets <> .

Fonctions génériques

Pour les fonctions , cet espace réservé est placé après le nom de la fonction:

```
/// 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
}
```

Dans ce cas, le générique est T Lorsque vous appelez la fonction, Swift peut déduire le type de T pour vous (car il agit simplement comme un espace réservé pour un type réel).

```
let randomOutput = pickRandom(5, 7) // returns an Int (that's either 5 or 7)
```

Nous passons ici deux entiers à la fonction. Par conséquent, Swift en déduit T == Int - la signature de la fonction est donc supposée être (Int, Int) -> Int .

En raison de la sécurité de type forte offerte par les génériques, les arguments et le retour de la fonction doivent être du *même* type. Par conséquent, les éléments suivants ne seront pas compilés:

```
struct Foo {}

let foo = Foo()

let randomOutput = pickRandom(foo, 5) // error: cannot convert value of type 'Int' to expected
argument type 'Foo'
```

Types génériques

Pour pouvoir utiliser des génériques avec des classes , des structures ou des énumérations , vous pouvez définir l'espace réservé générique après le nom du type.

```
class Bar<T> {
    var baz : T

    init(baz:T) {
        self.baz = baz
    }
}
```

Ce paramètre générique nécessitera un type lorsque vous utiliserez la classe Bar . Dans ce cas, il peut être déduit de l'initialiseur init(baz:T) .

```
let bar = Bar(baz: "a string") // bar's type is Bar<String>
```

Ici, l'espace réservé générique T est supposé être de type String , créant ainsi une instance Bar<String> . Vous pouvez également spécifier le type explicitement:

```
let bar = Bar<String>(baz: "a string")
```

Lorsqu'il est utilisé avec un type, l'espace réservé générique donné conservera son type pendant toute la durée de vie de l'instance donnée et ne peut pas être modifié après l'initialisation. Par conséquent, lorsque vous accédez à la propriété baz , il sera toujours de type String pour cette instance donnée.

```
let str = bar.baz // of type String
```

Passage de types génériques

Lorsque vous transmettez des types génériques, vous devez, dans la plupart des cas, être explicite sur le type générique que vous attendez. Par exemple, comme entrée de fonction:

```
func takeABarInt(bar:Bar<Int>) {
    ...
}
```

Cette fonction n'acceptera qu'une Bar<Int> . Si vous tentez de transmettre une instance de la Bar où le type générique de l'espace réservé n'est pas Int vous Int une erreur de compilation.

Nom générique d'espace réservé

Les noms génériques d'espace réservé ne se limitent pas à des lettres simples. Si un espace réservé donné représente un concept significatif, vous devez lui donner un nom descriptif. Par exemple, le Array de Swift a un espace réservé générique appelé Element , qui définit le type d'élément d'une instance de Array donnée.

```
public struct Array<Element> : RandomAccessCollection, MutableCollection {
    ...
}
```

Exemples de classes génériques

Une classe générique avec le paramètre type 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
    }
}
```

Nous pouvons maintenant créer de nouveaux objets en utilisant un paramètre de type

```
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
```

Les génériques peuvent également être créés avec plusieurs paramètres de type

```
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}
}
```

Et utilisé de la même manière

```
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
```

Héritage de classe générique

Les classes génériques peuvent être héritées:

```
// 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
    }

}
```

Utilisation de génériques pour simplifier les fonctions de tableau

Une fonction qui étend la fonctionnalité du tableau en créant une fonction de suppression orientée objet.

```
// 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")
        }
    }
}
```

Usage

```
var myArray = [1,2,3]
print(myArray)

// Prints [1,2,3]
```

Utilisez la fonction pour supprimer un élément sans avoir besoin d'index. Il suffit de passer l'objet à supprimer.

```
myArray.remove(2)
print(myArray)

// Prints [1,3]
```

Utiliser des génériques pour améliorer la sécurité des types

Prenons cet exemple sans utiliser de génériques

```
protocol JSONDecodable {
    static func from(_ json: [String: Any]) -> Any?
}
```

La déclaration de protocole semble correcte, sauf si vous l'utilisez réellement.

```
let myTestObject = TestObject.from(myJson) as? TestObject
```

Pourquoi devez-vous lancer le résultat dans TestObject ? En raison du type de retour Any dans la

déclaration de protocole.

En utilisant des génériques, vous pouvez éviter ce problème qui peut provoquer des erreurs d'exécution (et nous ne voulons pas les avoir!)

```
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?`
```

Contraintes de type avancées

Il est possible de spécifier plusieurs contraintes de type pour les génériques en utilisant la clause 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.")
    }
}
```

Il est également valable d'écrire la clause where après la liste des arguments:

```
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.")
    }
}
```

Les extensions peuvent être limitées à des types satisfaisant aux conditions. La fonction est uniquement disponible pour les instances qui satisfont aux conditions de type:

```
// "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")
        }
    }
}
```

Lire Génériques en ligne: https://riptutorial.com/fr/swift/topic/774/generiques

Chapitre 29: Gestion de la mémoire

Introduction

Cette rubrique décrit comment et quand le runtime Swift alloue de la mémoire pour les structures de données d'application et quand cette mémoire doit être récupérée. Par défaut, les instances de la classe de sauvegarde de la mémoire sont gérées via le comptage des références. Les structures passent toujours par la copie. Pour désactiver le schéma de gestion de la mémoire intégrée, vous pouvez utiliser la structure [Unmanaged] [1]. [1]: https://developer.apple.com/reference/swift/unmanaged

Remarques

Quand utiliser le mot-clé faible:

Le mot-clé weak doit être utilisé si un objet référencé peut être désalloué pendant la durée de vie de l'objet contenant la référence.

Quand utiliser le mot-clé sans propriétaire:

Le unowned clé unowned propriétaire doit être utilisé si un objet référencé n'est pas censé être désalloué pendant la durée de vie de l'objet contenant la référence.

Pièges

Une erreur fréquente est d'oublier de créer des références à des objets, qui doivent survivre à la fin d'une fonction, comme les gestionnaires de lieu, les gestionnaires de mouvement, etc.

Exemple:

```
class A : CLLocationManagerDelegate
{
    init()
    {
        let locationManager = CLLocationManager()
        locationManager.delegate = self
        locationManager.startLocationUpdates()
    }
}
```

Cet exemple ne fonctionnera pas correctement, car le gestionnaire d'emplacement est désalloué après le retour de l'initialiseur. La solution appropriée consiste à créer une référence forte en tant que variable d’instance:

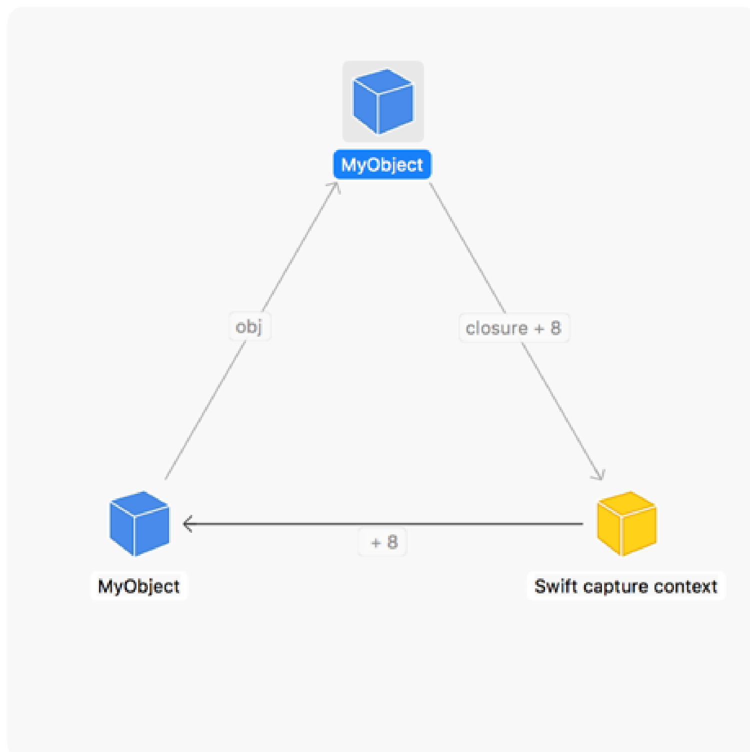
```
class A : CLLocationManagerDelegate
{
    let locationManager:CLLocationManager

    init()
    {
        locationManager = CLLocationManager()
        locationManager.delegate = self
        locationManager.startLocationUpdates()
    }
}
```

Examples

Cycles de référence et références faibles

Un *cycle de référence* (ou *cycle de conservation*) est nommé ainsi car il indique un cycle dans le graphe d'objet :



Chaque flèche indique un objet en retenant un autre (une référence forte). À moins que le cycle ne soit rompu, la mémoire de ces objets **ne** sera **jamais libérée** .

Un cycle de conservation est créé lorsque deux instances de classes se réfèrent l'une à l'autre:

```
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
```

Les deux instances resteront en vie jusqu'à la fin du programme. Ceci est un cycle de rétention.

Références faibles

Pour éviter les cycles de rétention, utilisez le mot-clé weak ou sans unowned lors de la création de références pour les cycles de retenue.

```
class B { weak var a: A? = nil }
```

Des références faibles ou non acquises n'augmenteront pas le nombre de références d'une instance. Ces références ne contribuent pas à conserver les cycles. La référence faible **devient nil** lorsque l'objet référencé est désalloué.

```
a.b = b  // a retains b
b.a = a  // b holds a weak reference to a -- not a reference cycle
```

Lorsque vous travaillez avec des fermetures, vous pouvez également utiliser weak listes de capture weak ou unowned .

Gestion de la mémoire manuelle

Lors de l’interfaçage avec les API C, il est possible de désactiver le compteur de référence Swift. Cela se fait avec des objets non gérés.

Si vous devez fournir un pointeur de type puned à une fonction C, utilisez la méthode toOpaque de la structure Unmanaged pour obtenir un pointeur brut et fromOpaque pour récupérer l'instance d'origine:

```
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()
}
```

Notez que si vous utilisez passUnretained et ses homologues, il est nécessaire de prendre toutes les précautions comme avec les références non- unowned .

Pour interagir avec les API Objective-C héritées, il peut être nécessaire d'affecter manuellement le nombre de références d'un objet donné. Pour cela, Unmanaged a retain et release méthodes respectives. Néanmoins, il est passRetained utiliser passRetained et takeRetainedValue , qui effectuent des conservations avant de renvoyer le résultat:

```
func preferredFilenameExtension(for uti: String) -> String! {
  let result = UTTypeCopyPreferredTagWithClass(uti, kUTTagClassFilenameExtension)
  guard result != nil else { return nil }

  return result!.takeRetainedValue() as String
}
```

Ces solutions doivent toujours être le dernier recours et les API natives doivent toujours être préférées.

Lire Gestion de la mémoire en ligne: https://riptutorial.com/fr/swift/topic/745/gestion-de-la-memoire

Chapitre 30: Gestionnaire d'achèvement

Introduction

Pratiquement toutes les applications utilisent des fonctions asynchrones pour empêcher le code de bloquer le thread principal.

Examples

Gestionnaire d'achèvement sans argument d'entrée

```
func sampleWithCompletion(completion:@escaping (()-> ())){
    let delayInSeconds = 1.0
    DispatchQueue.main.asyncAfter(deadline: DispatchTime.now() + delayInSeconds) {

        completion()

    }
}

//Call the function
sampleWithCompletion {
    print("after one second")
}
```

Gestionnaire d'achèvement avec un argument d'entrée

```
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

    }
}
```

Lire Gestionnaire d'achèvement en ligne:
https://riptutorial.com/fr/swift/topic/9378/gestionnaire-d-achevement

Chapitre 31: Gestionnaire de paquets rapide

Examples

Création et utilisation d'un simple package Swift

Pour créer un package Swift, ouvrez un terminal puis créez un dossier vide:

```
mkdir AwesomeProject
cd AwesomeProject
```

Et initiez un dépôt Git:

```
git init
```

Ensuite, créez le paquet lui-même. On pourrait créer la structure du paquet manuellement mais il existe un moyen simple d'utiliser la commande CLI.

Si vous voulez faire un exécutable:

```
swift package init --type executable
```

Plusieurs fichiers seront générés. Parmi eux, *main.swift* sera le point d'entrée de votre application.

Si vous voulez créer une bibliothèque:

```
swift package init --type library
```

Le fichier *AwesomeProject.swift* généré sera utilisé comme fichier principal pour cette bibliothèque.

Dans les deux cas, vous pouvez ajouter d'autres fichiers Swift dans le dossier *Sources* (les règles habituelles pour le contrôle d'accès s'appliquent).

Le fichier *Package.swift* lui-même sera automatiquement *rempli* avec ce contenu:

```
import PackageDescription

let package = Package(
    name: "AwesomeProject"
)
```

La version du paquet est faite avec les tags Git:

```
git tag '1.0.0'
```

Une fois transféré dans un dépôt Git distant ou local, votre package sera disponible pour d'autres projets.

Votre paquet est maintenant prêt à être compilé:

```
swift build
```

Le projet compilé sera disponible dans le dossier *.build / debug* .

Votre propre package peut également résoudre les dépendances vers d'autres packages. Par exemple, si vous souhaitez inclure "SomeOtherPackage" dans votre propre projet, modifiez votre fichier *Package.swift* pour inclure la dépendance:

```
import PackageDescription

let package = Package(
    name: "AwesomeProject",
    targets: [],
    dependencies: [
        .Package(url: "https://github.com/someUser/SomeOtherPackage.git",
                 majorVersion: 1),
    ]
)
```

Ensuite, reconstruisez votre projet: le gestionnaire de paquets Swift va automatiquement résoudre, télécharger et créer les dépendances.

Lire Gestionnaire de paquets rapide en ligne:
https://riptutorial.com/fr/swift/topic/5144/gestionnaire-de-paquets-rapide

Chapitre 32: Hachage cryptographique

Examples

MD2, MD4, MD5, SHA1, SHA224, SHA256, SHA384, SHA512 (Swift 3)

> Ces fonctions hacheront les entrées String ou Data avec l'un des huit algorithmes de hachage cryptographiques.

Le paramètre name spécifie le nom de la fonction de hachage en tant que chaîne.
Les fonctions prises en charge sont MD2, MD4, MD5, SHA1, SHA224, SHA256, SHA384 et SHA512

Cet exemple nécessite Common Crypto
Il est nécessaire d'avoir un en-tête de pontage pour le projet:
#import <CommonCrypto/CommonCrypto.h>
Ajoutez le Security.framework au projet.

Cette fonction prend un nom de hachage et des données à hacher et retourne une donnée:

```
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
}
```

Cette fonction prend un nom de hachage et une chaîne à hacher et retourne un Data:

```
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)
}
```

Exemples:

```
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)")
```

Sortie:

```
clearString: clearData0123456
clearData: <636c6561 72446174 61303132 33343536>

hashSHA256: <aabc766b 6b357564 e41f4f91 2d494bcc bfa16924 b574abbd ba9e3e9d a0c8920a>
hashMD5: <4df665f7 b94aea69 695b0e7b baf9e9d6>
```

HMAC avec MD5, SHA1, SHA224, SHA256, SHA384, SHA512 (Swift 3)

> Ces fonctions hacheront les entrées String ou Data avec l'un des huit algorithmes de hachage cryptographiques.

Le paramètre name spécifie le nom de la fonction de hachage en tant que chaîne. Les fonctions prises en charge sont MD5, SHA1, SHA224, SHA256, SHA384 et SHA512.

Cet exemple nécessite Common Crypto
Il est nécessaire d'avoir un en-tête de pontage pour le projet:
#import <CommonCrypto/CommonCrypto.h>
Ajoutez le Security.framework au projet.

Ces fonctions prennent un nom de hachage, un message à hacher, une clé et renvoient un résumé:

```
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)
}
```

// Exemples

```
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)")
```

Sortie:

```
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>
```

Lire Hachage cryptographique en ligne: https://riptutorial.com/fr/swift/topic/7885/hachage-cryptographique

Chapitre 33: Initialiseurs

Examples

Définition des valeurs de propriété par défaut

Vous pouvez utiliser un initialiseur pour définir les valeurs de propriété par défaut:

```
struct Example {
    var upvotes: Int
    init() {
        upvotes = 42
    }
}
let myExample = Example() // call the initializer
print(myExample.upvotes) // prints: 42
```

Ou, spécifiez les valeurs de propriété par défaut dans le cadre de la déclaration de la propriété:

```
struct Example {
    var upvotes = 42 // the type 'Int' is inferred here
}
```

Les classes et les structures **doivent** définir toutes les propriétés stockées sur une valeur initiale appropriée au moment de la création d'une instance. Cet exemple ne sera pas compilé, car l'initialiseur n'a pas donné de valeur initiale pour les downvotes :

```
struct Example {
    var upvotes: Int
    var downvotes: Int
    init() {
         upvotes = 0
    } // error: Return from initializer without initializing all stored properties
}
```

Personnalisation de l'initialisation avec les paramètres

```
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
```

Notez que vous ne pouvez pas omettre les étiquettes de paramètre:

```
let myBadDistance = MetricDistance(42) // error: argument labels do not match any available
overloads
```

Pour permettre l’omission des étiquettes de paramètres, utilisez un trait de soulignement _ comme étiquette:

```
struct MetricDistance {
    var distanceInMeters: Double
    init(_ meters: Double) {
        distanceInMeters = meters
    }
}
let myDistance = MetricDistance(42) // distanceInMeters = 42
```

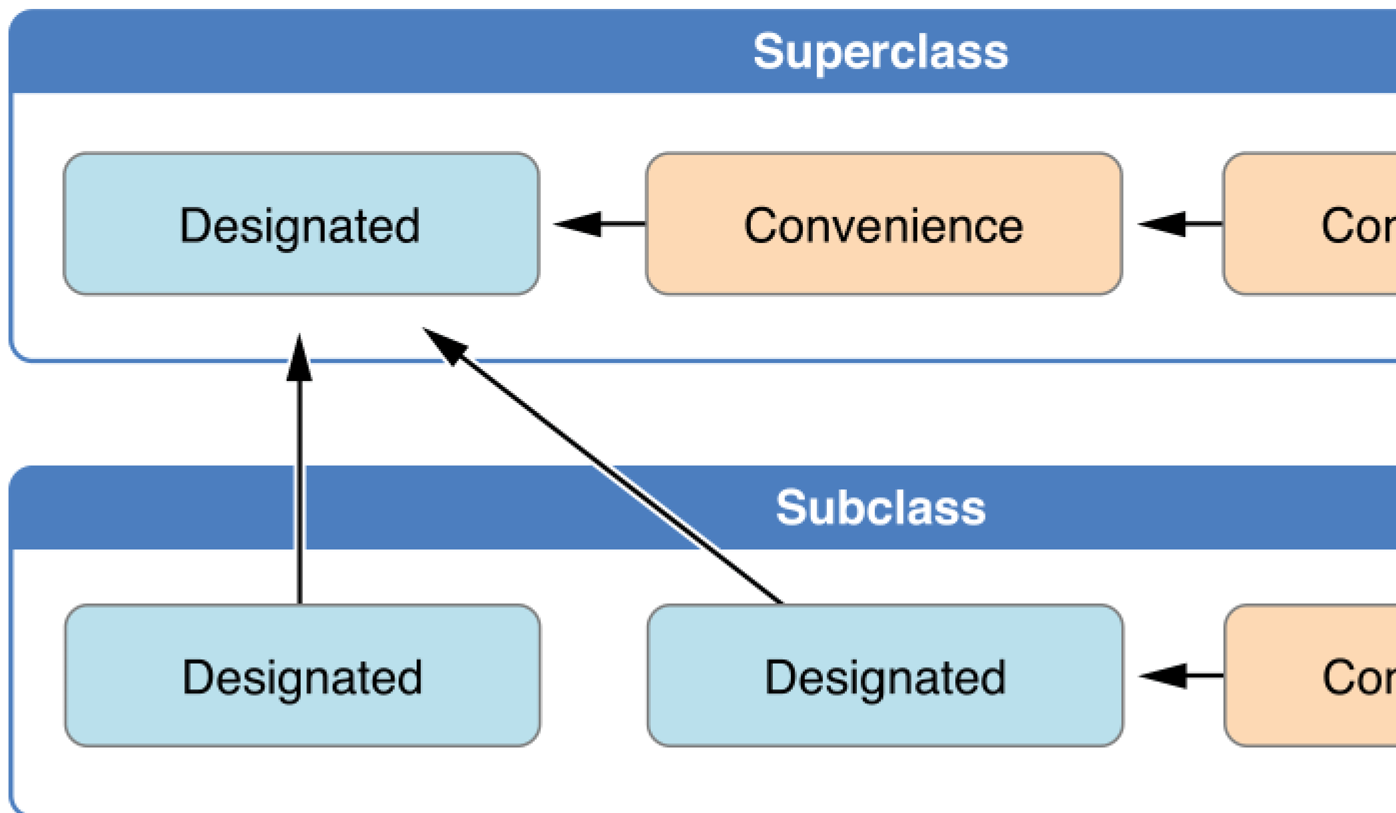
Si vos étiquettes d'argument partagent des noms avec une ou plusieurs propriétés, utilisez self pour définir explicitement les valeurs de propriété:

```
struct Color {
    var red, green, blue: Double
    init(red: Double, green: Double, blue: Double) {
        self.red = red
        self.green = green
        self.blue = blue
    }
}
```

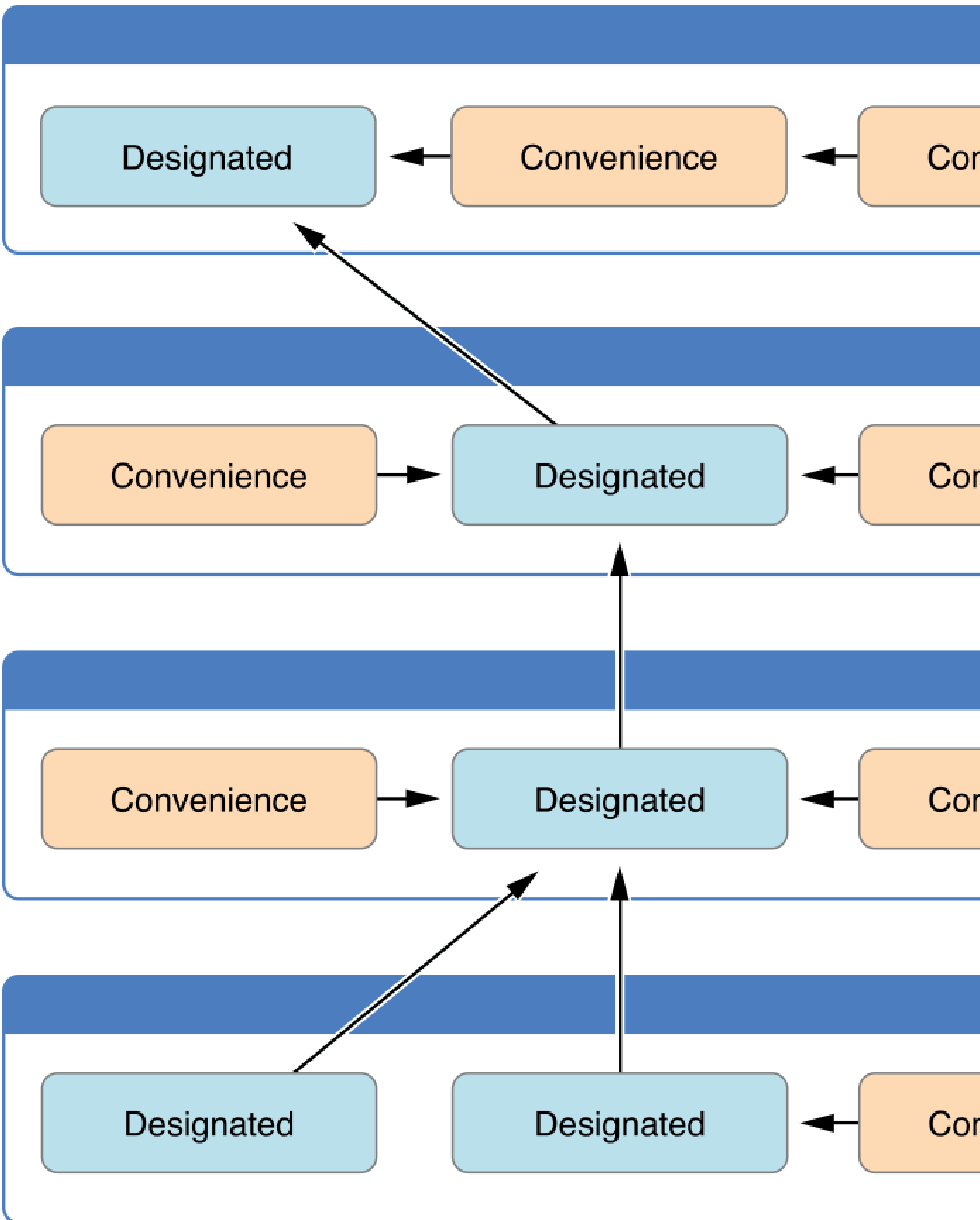
Commencement init

Les classes Swift prennent en charge plusieurs manières d'être initialisées. Suivant les spécifications d'Apple, ces 3 règles doivent être respectées:

1. Un initialiseur désigné doit appeler un initialiseur désigné à partir de sa super-classe immédiate.



2. Un initialiseur de commodité doit appeler un autre initialiseur de la même classe.
3. Un initialiseur de commodité doit finalement appeler un initialiseur désigné.




```
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)
    }
}
```

Initialiseur désigné

```
let c = Foo(someString: "Some string", someValue: 10, someBool: true)
```

Commodité init ()

```
let a = Foo()
```

Init initiale (otherString: String)

```
let b = Foo(otherString: "Some string")
```

Initialisateur désigné (appellera l'initialiseur désigné de la superclasse)

```
let d = Baz(someFloat: 3)
```

Commodité init ()

```
let e = Baz()
```

Source d'image: La langue de programmation Swift

Lanceur Jetable

Utilisation de la gestion des erreurs pour créer un initialiseur Struct (ou de classe) en tant qu'initialiseur pouvant être lancé:

Exemple de gestion des erreurs enum:

```
enum ValidationError: Error {
    case invalid
}
```

Vous pouvez utiliser l'énumération de gestion des erreurs pour vérifier le paramètre de la structure (ou de la classe) répondant aux exigences requises

```
struct User {
    let name: String

    init(name: String?) throws {

        guard let name = name else {
           ValidationError.invalid
        }

        self.name = name
    }
}
```

Maintenant, vous pouvez utiliser l'initialiseur lancable par:

```
do {
   let user = try User(name: "Sample name")

   // success
}
catch ValidationError.invalid {
     // handle error
}
```

Lire Initialiseurs en ligne: https://riptutorial.com/fr/swift/topic/1778/initialiseurs

Chapitre 34: Injection de dépendance

Examples

Injection de dépendance avec les contrôleurs de vue

Injection Dependenct Intro

Une application est composée de nombreux objets qui collaborent entre eux. Les objets dépendent généralement d'autres objets pour effectuer certaines tâches. Lorsqu'un objet est responsable de référencer ses propres dépendances, il en résulte un code hautement couplé, difficile à tester et difficile à changer.

L'injection de dépendance est un modèle de conception logicielle qui implémente l'inversion du contrôle pour résoudre les dépendances. Une injection consiste à transmettre une dépendance à un objet dépendant qui l'utilise. Cela permet de séparer les dépendances du client du comportement du client, ce qui permet à l'application d'être couplée de manière souple.

Ne pas confondre avec la définition ci-dessus - une injection de dépendance signifie simplement donner à un objet ses variables d'instance.

C'est aussi simple que cela, mais il offre de nombreux avantages:

- plus facile de tester votre code (en utilisant des tests automatisés comme les tests d'unité et d'interface utilisateur)
- lorsqu'il est utilisé en tandem avec une programmation orientée protocole, il est facile de modifier l'implémentation d'une certaine classe - plus facile à restructurer
- cela rend le code plus modulaire et réutilisable

Il existe trois méthodes les plus couramment utilisées pour l’injection de dépendance (DI) dans une application:

1. Injection d'initialisation
2. Injection de propriété
3. Utiliser des frameworks DI tiers (comme Swinject, Cleanse, Dip ou Typhoon)

Il y a un article intéressant avec des liens vers d'autres articles sur l'injection de dépendances, alors jetez-y un coup d'œil si vous voulez approfondir le principe du DI et de l'inversion du contrôle.

Montrons comment utiliser DI avec View Controllers - une tâche quotidienne pour un développeur iOS moyen.

Exemple sans DI

Nous aurons deux contrôleurs de vue: **LoginViewController** et **TimelineViewController** . LoginViewController est utilisé pour se connecter et en cas de succès, il basculera vers le TimelineViewController. Les deux contrôleurs de vue dépendent du **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
    }
}
```

Cet exemple est très simple, mais supposons que vous ayez 10 ou 15 contrôleurs de vue différents et que certains d'entre eux dépendent également du FirebaseNetworkService. À un moment donné, vous souhaitez modifier Firebase en tant que service backend avec le service back-end de votre entreprise. Pour ce faire, vous devez passer par chaque contrôleur de vue et modifier FirebaseNetworkService avec CompanyNetworkService. Et si certaines des méthodes de CompanyNetworkService ont changé, vous aurez beaucoup de travail à faire.

Les tests unitaires et d'interface utilisateur ne sont pas la portée de cet exemple, mais si vous vouliez unifier les contrôleurs de vue de test avec des dépendances étroitement couplées, vous auriez vraiment du mal à le faire.

Réécrivons cet exemple et injectons le service réseau dans nos contrôleurs de vue.

Exemple avec injection de dépendance

Pour tirer le meilleur parti de l'injection de dépendances, définissons les fonctionnalités du service réseau dans un protocole. De cette façon, les contrôleurs de vue dépendant d'un service réseau n'auront même pas besoin d'en connaître la mise en œuvre réelle.

```
protocol NetworkService {
    func loginUser(username: String, passwordHash: String)
    func logutCurrentUser()
}
```

Ajoutez une implémentation du protocole NetworkService:

```
class FirebaseNetworkServiceImpl: NetworkService {
    func loginUser(username: String, passwordHash: String) {
        // Firebase implementation
    }
```

```
    func logutCurrentUser() {
        // Firebase implementation
    }
}
```

Modifions LoginViewController et TimelineViewController pour utiliser le nouveau protocole NetworkService au lieu 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()
    }
}
```

Maintenant, la question est la suivante: comment pouvons-nous injecter l'implémentation NetworkService correcte dans LoginViewController et TimelineViewController?

Puisque LoginViewController est le contrôleur de vue de départ et qu'il s'affiche à chaque démarrage de l'application, nous pouvons injecter toutes les dépendances dans **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
}
```

Dans AppDelegate, nous prenons simplement la référence au premier contrôleur de vue (LoginViewController) et injectons l'implémentation NetworkService à l'aide de la méthode d'injection de propriété.

Maintenant, la tâche suivante consiste à injecter l'implémentation NetworkService dans le TimelineViewController. Le moyen le plus simple consiste à le faire lorsque LoginViewController est en transition vers TimlineViewController.

Nous allons ajouter le code d'injection dans la méthode prepareForSegue dans LoginViewController (si vous utilisez une approche différente pour naviguer dans les contrôleurs de vues, placez-y le code d'injection).

Notre classe LoginViewController ressemble à ceci maintenant:

```
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
            }
        }
    }
}
```

Nous avons terminé et c'est aussi simple que cela.

Maintenant, imaginons que nous souhaitons faire passer notre implémentation NetworkService de Firebase à l'implémentation backend de notre société personnalisée. Tout ce que nous aurions à faire est de:

Ajoutez une nouvelle classe d'implémentation NetworkService:

```
class CompanyNetworkServiceImpl: NetworkService {
    func loginUser(username: String, passwordHash: String) {
        // Company API implementation
    }

    func logutCurrentUser() {
        // Company API implementation
    }
}
```

Basculez le FirebaseNetworkServiceImpl avec la nouvelle implémentation dans 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
    }
```

Ça y est, nous avons changé toute l'implémentation sous-jacente du protocole NetworkService sans même toucher à LoginViewController ou TimelineViewController.

Comme il s'agit d'un exemple simple, vous pourriez ne pas voir tous les avantages dès maintenant, mais si vous essayez d'utiliser DI dans vos projets, vous en verrez les avantages et vous utiliserez toujours l'injection de dépendance.

Types d'injection de dépendance

Cet exemple démontrera comment utiliser modèle de conception d'injection de dépendance **(DI)** à Swift en utilisant ces méthodes:

1. **Initializer Injection** (le terme approprié est Constructor Injection, mais étant donné que les initialiseurs de Swift sont appelés des initialiseurs d'injection)
2. **Injection de propriété**
3. **Méthode injection**

Exemple d'installation sans 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?
}
```

Injection de dépendance d'initialisation

Comme son nom l'indique, toutes les dépendances sont injectées via l'initialiseur de classe. Pour injecter des dépendances via l'initialiseur, nous allons ajouter l'initialiseur à la classe Train .

La classe de train ressemble maintenant à ceci:

```
class Train {
    let engine: Engine?
    var mainCar: TrainCar?

    init(engine: Engine) {
        self.engine = engine
    }
}
```

Lorsque nous voulons créer une instance de la classe Train, nous utiliserons l'initialiseur pour injecter une implémentation spécifique du moteur:

```
let train = Train(engine: TrainEngine())
```

REMARQUE: Le principal avantage de l'injection de l'initialiseur par rapport à l'injection de propriété est que nous pouvons définir la variable en tant que variable privée ou même en faire une constante avec le mot clé let (comme nous l'avons fait dans notre exemple). De cette façon, nous pouvons nous assurer que personne ne peut y accéder ou le modifier.

Propriétés Injection de dépendance

DI utilisant des propriétés est encore plus simple que l'utilisation d'un initialiseur. Nous allons injecter une dépendance PassengerCar à l'objet de train que nous avons déjà créé en utilisant les propriétés DI:

```
train.mainCar = PassengerCar()
```

C'est tout. mainCar notre train est maintenant une instance PassengerCar .

Méthode d'injection de dépendance

Ce type d'injection de dépendances est un peu différent des deux précédentes car il n'affectera pas l'objet entier, mais n'injectera qu'une dépendance à utiliser dans le cadre d'une méthode spécifique. Lorsqu'une dépendance n'est utilisée que dans une seule méthode, il n'est généralement pas bon de la rendre dépendante de l'objet entier. Ajoutons une nouvelle méthode à la classe Train:

```
func reparkCar(trainCar: TrainCar) {
    trainCar.attachCar(attach: true)
    engine?.startEngine()
    engine?.stopEngine()
    trainCar.attachCar(attach: false)
```

```
}
```

Maintenant, si nous appelons la méthode de classe du nouveau Train, nous allons injecter le TrainCar utilisant l'injection de dépendance de méthode.

```
train.reparkCar(trainCar: RestaurantCar())
```

Lire Injection de dépendance en ligne: https://riptutorial.com/fr/swift/topic/8198/injection-de-dependance

Chapitre 35: La déclaration différée

Examples

Quand utiliser une déclaration différée

Une instruction de defer consiste en un bloc de code, qui sera exécuté lors du retour d'une fonction et qui devrait être utilisé pour le nettoyage.

Comme les déclarations de guard de Swift encouragent un style de retour précoce, de nombreux chemins possibles pour un retour peuvent exister. Une defer déclaration fournit le code de nettoyage, qui ne puis pas besoin d'être répété à chaque fois.

Il peut également gagner du temps lors du débogage et du profilage, car les fuites de mémoire et les ressources ouvertes non utilisées en raison d'un nettoyage oublié peuvent être évitées.

Il peut être utilisé pour désallouer un tampon à la fin d'une fonction:

```
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.
```

Il peut également être utilisé pour fermer des ressources à la fin d'une fonction:

```
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
```

Quand ne pas utiliser une déclaration différée

Lorsque vous utilisez une instruction de report, assurez-vous que le code reste lisible et que l'ordre d'exécution reste clair. Par exemple, l'utilisation suivante de l'instruction defer rend l'ordre d'exécution et la fonction du code difficiles à comprendre.

```
postfix func ++ (inout value: Int) -> Int {
    defer { value += 1 } // do NOT do this!
    return value
}
```

Lire La déclaration différée en ligne: https://riptutorial.com/fr/swift/topic/4932/la-declaration-differee

Chapitre 36: La gestion des erreurs

Remarques

Pour plus d'informations sur les erreurs, voir le langage de programmation Swift .

Examples

Bases de traitement des erreurs

Les fonctions dans Swift peuvent renvoyer des valeurs, **lancer des erreurs** ou les deux:

```
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
```

Toute valeur conforme au protocole ErrorType (y compris les objets NSError) peut être considérée comme une erreur. Les énumérations constituent un moyen pratique de définir des erreurs personnalisées:

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)
}
```

Une erreur indique une défaillance non fatale au cours de l'exécution du programme et est traitée avec les constructions de contrôle-flux spécialisées do / catch , throw et 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
    }
}
```

Les erreurs peuvent être détectées avec 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)")
}
```

Toute fonction pouvant générer une erreur ***doit*** être appelée en utilisant try , try? ou 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)
```

Attraper différents types d'erreur

Créons notre propre type d'erreur pour cet exemple.

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 syntaxe Do-Catch permet de détecter une erreur générée et crée *automatiquement* une error nommée constante disponible dans le bloc catch :

```
do {
    try throwing()
} catch {
    print(error)
}
```

Vous pouvez également déclarer une variable vous-même:

```
do {
    try throwing()
} catch let oops {
    print(oops)
}
```

Il est également possible de chaîner différents énoncés de catch . Ceci est pratique si plusieurs types d'erreurs peuvent être générés dans le bloc Do.

Ici, le Do-Catch tentera d'abord de CustomError l'erreur en CustomError , puis en NSError si le type personnalisé n'a pas été NSError .

2.2

```
do {
    try somethingMayThrow()
} catch let custom as CustomError {
    print(custom)
} catch let error as NSError {
    print(error)
}
```

3.0

Dans Swift 3, pas besoin de baisser explicitement vers NSError.

```
do {
    try somethingMayThrow()
} catch let custom as CustomError {
    print(custom)
} catch {
    print(error)
}
```

Modèle de capture et de commutation pour la gestion explicite des erreurs

```
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
        }
    }

}
```

Dans la classe client:

```
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
    }
}
```

Désactivation de la propagation des erreurs

Les créateurs de Swift ont mis beaucoup d'attention à ce que le langage soit expressif et que la gestion des erreurs soit tout à fait expressive. Si vous essayez d'appeler une fonction pouvant générer une erreur, l'appel de la fonction doit être précédé du mot-clé try. Le mot clé try n'est pas magique. Tout ce qu'il fait, c'est de sensibiliser le développeur à la capacité de lancement de la fonction.

Par exemple, le code suivant utilise une fonction loadImage (atPath :), qui charge la ressource image sur un chemin donné ou génère une erreur si l'image ne peut pas être chargée. Dans ce cas, étant donné que l'image est livrée avec l'application, aucune erreur ne sera générée lors de l'exécution. Par conséquent, il est recommandé de désactiver la propagation des erreurs.

```
let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```

Créer une erreur personnalisée avec une description localisée

Créer un enum d'erreurs personnalisées

```
enum RegistrationError: Error {
    case invalidEmail
    case invalidPassword
    case invalidPhoneNumber
}
```

Créer une extension de RegistrationError pour gérer la description localisée.

```
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")
        }
    }
}
```

Erreur de manipulation:

```
let error: Error = RegistrationError.invalidEmail
print(error.localizedDescription)
```

Lire La gestion des erreurs en ligne: https://riptutorial.com/fr/swift/topic/283/la-gestion-des-erreurs

Chapitre 37: Lecture et écriture JSON

Syntaxe

- NSJSONSerialization.JSONObjectWithData (jsonData, options: NSJSONReadingOptions) // Renvoie un objet à partir de jsonData. Cette méthode échoue.
- NSJSONSerialization.dataWithJSONObject (jsonObject, options: NSJSONWritingOptions) // Renvoie NSData à partir d'un objet JSON. Transmettez NSJSONWritingOptions.PrettyPrinted dans les options pour une sortie plus lisible.

Examples

Sérialisation JSON, encodage et décodage avec Apple Foundation et la bibliothèque Swift Standard

La classe JSONSerialization est intégrée au framework Foundation d'Apple.

2.2

Lire JSON

La fonction JSONObjectWithData prend NSData et renvoie AnyObject . Vous pouvez utiliser as? pour convertir le résultat à votre type attendu.

```
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)")
}
```

Vous pouvez passer des options: .AllowFragments au lieu des options: [] pour autoriser la lecture de JSON lorsque l'objet de niveau supérieur n'est pas un tableau ou un dictionnaire.

Ecrire JSON

L'appel de dataWithJSONObject convertit un objet compatible JSON (tableaux imbriqués ou dictionnaires avec des chaînes, des nombres et NSNull) en NSData brut codé en 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)")
}
```

Vous pouvez passer des options: .PrettyPrinted au lieu des options: [] pour une jolie impression.

3.0

Même comportement dans Swift 3 mais avec une syntaxe différente.

```
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)")
}
```

> Remarque: La section suivante est actuellement disponible uniquement dans **Swift 4.0** et versions ultérieures.

À partir de Swift 4.0, la bibliothèque standard Swift inclut les protocoles Encodable et Decodable pour définir une approche normalisée du codage et du décodage des données. L'adoption de ces protocoles permettra aux implémentations des protocoles Encoder et Decoder prendre vos données et de les encoder ou les décoder depuis et vers une représentation externe telle que JSON. La conformité au protocole Codable combine les protocoles Encodable et Decodable . C'est maintenant le moyen recommandé de gérer JSON dans votre programme.

Encoder et décoder automatiquement

La méthode la plus simple pour rendre un type codable consiste à déclarer ses propriétés comme des types déjà Codable . Ces types incluent les types de bibliothèque standard tels que String , Int et Double ; et les types de Fondation tels que Date , Data et URL . Si les propriétés d'un type sont codables, le type lui-même se conformera automatiquement à Codable en déclarant simplement la conformité.

Prenons l'exemple suivant, dans lequel la structure du Book est conforme à Codable .

```
struct Book: Codable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

> Notez que les collections standard telles que Array et Dictionary sont conformes à Codable si elles contiennent des types codables.

En adoptant Codable , la structure Book peut maintenant être codée et décodée à partir de JSON en utilisant les classes Apple Foundation JSONEncoder et JSONDecoder , même si Book lui-même ne contient aucun code pour gérer spécifiquement JSON. Les codeurs et décodeurs personnalisés peuvent également être écrits en respectant respectivement les protocoles de Encoder et de Decoder .

Encoder les données 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)
```

> Définissez encoder.outputFormatting = .prettyPrinted pour faciliter la lecture. ## Décoder à partir de données JSON

Décoder à partir de données 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)
```

> Dans l'exemple ci-dessus, Book.self informe le décodeur du type auquel le JSON doit être décodé.

Encodage ou décodage exclusivement

Parfois, vous n'avez peut-être pas besoin de données codables et décodables, par exemple lorsque vous avez uniquement besoin de lire des données JSON à partir d'une API ou que votre programme ne soumet que des données JSON à une API.

Si vous avez l'intention d'écrire uniquement des données JSON, conformez-vous à Encodable .

```
struct Book: Encodable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

Si vous souhaitez uniquement lire les données JSON, conformez-vous à Decodable .

```
struct Book: Decodable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

Utilisation de noms de clé personnalisés

Les API utilisent fréquemment des conventions de dénomination autres que le cas camel standard Swift, tel que le cas du serpent. Cela peut devenir un problème quand il s'agit de décoder JSON, car par défaut les clés JSON doivent s'aligner exactement sur les noms de propriété de votre

type. Pour gérer ces scénarios, vous pouvez créer des clés personnalisées pour votre type à l'aide du protocole CodingKey .

```
struct Book: Codable {
    // ...
    enum CodingKeys: String, CodingKey {
        case title
        case authors
        case publicationDate = "publication_date"
    }
}
```

CodingKeys sont générés automatiquement pour les types qui adoptent le protocole Codable , mais en créant notre propre implémentation dans l'exemple ci-dessus, nous Codable notre décodeur à correspondre à la publicationDate cas camel avec le cas de publication_date tel qu'il est fourni par l'API.

SwiftyJSON

SwiftyJSON est un framework Swift conçu pour supprimer le chaînage facultatif dans la sérialisation JSON normale.

Vous pouvez le télécharger ici: https://github.com/SwiftyJSON/SwiftyJSON

Sans SwiftyJSON, votre code ressemblerait à ceci pour trouver le nom du premier livre dans un objet 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
}
```

Dans SwiftyJSON, ceci est extrêmement simplifié:

```
let json = JSON(data: data)
if let bookName = json[0]["book"]["name"].string {
    //We can now use the book name
}
```

Cela supprime la nécessité de vérifier chaque champ, car il renverra zéro si l'un d'entre eux est invalide.

Pour utiliser SwiftyJSON, téléchargez la version correcte depuis le dépôt Git - il existe une branche pour Swift 3. Faites simplement glisser le "SwiftyJSON.swift" dans votre projet et importez-le dans votre classe:

```
import SwiftyJSON
```

Vous pouvez créer votre objet JSON en utilisant les deux initialiseurs suivants:

```
let jsonObject = JSON(data: dataObject)
```

ou

```
let jsonObject = JSON(jsonObject) //This could be a string in a JSON format for example
```

Pour accéder à vos données, utilisez les indices:

```
let firstObjectInAnArray = jsonObject[0]
let nameOfFirstObject = jsonObject[0]["name"]
```

Vous pouvez ensuite analyser votre valeur à un certain type de données, qui renverra une valeur facultative:

```
let nameOfFirstObject = jsonObject[0]["name"].string //This will return the name as a string

let nameOfFirstObject = jsonObject[0]["name"].double //This will return null
```

Vous pouvez également compiler vos chemins dans un tableau rapide:

```
let convolutedPath = jsonObject[0]["name"][2]["lastName"]["firstLetter"].string
```

Est le même que:

```
let convolutedPath = jsonObject[0, "name", 2, "lastName", "firstLetter"].string
```

SwiftyJSON a également des fonctionnalités pour imprimer ses propres erreurs:

```
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 vous devez écrire sur votre objet JSON, vous pouvez utiliser à nouveau les indices:

```
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 vous avez besoin de la chaîne d'origine pour JSON, par exemple si vous devez l'écrire dans un fichier, vous pouvez obtenir la valeur brute:

```
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 est une bibliothèque d'analyse JSON gérée par Big Nerd Ranch . Il présente trois avantages principaux:

1. Type Safety: vous aide à travailler avec l'envoi et la réception de JSON de manière à éviter les pannes à l'exécution.
2. Idiomatic: profite des génériques, des énumérations et des fonctionnalités de Swift, sans

documentation compliquée ni opérateurs personnalisés magiques.

3. Gestion des erreurs: Fournit des informations d'erreur informatives pour les erreurs JSON courantes.

Exemple de données JSON

Définissons quelques exemples de données JSON à utiliser avec ces exemples.


```
{
  "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)!
```

Désérialisation des données brutes

Pour désérialiser les données, nous initialisons un objet JSON puis accédons à une clé particulière.

```
do {
    let json = try JSON(data: jsonData)
    let success = try json.bool("success")
} catch {
    // do something with the error
}
```

Nous try ici parce que l'accès à json pour la clé "success" pourrait échouer - il pourrait ne pas exister, ou la valeur pourrait ne pas être un booléen.

Nous pouvons également spécifier un chemin d'accès aux éléments imbriqués dans la structure JSON. Le chemin est une liste de clés et d'indices séparés par des virgules qui décrivent le chemin d'accès à une valeur d'intérêt.

```
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
}
```

Désérialisation des modèles directement

JSON peut être directement analysé dans une classe de modèle qui implémente le protocole 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
}
```

Sérialisation des données brutes

Toute valeur JSON peut être sérialisée directement à NSData .

```
let success = JSON.Bool(false)
let data: NSData = try success.serialize()
```

Sérialisation de modèles directement

Toute classe de modèle qui implémente le protocole JSONEncodable peut être sérialisée directement dans 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()
```

Flèche

Arrow est une bibliothèque d'analyse JSON élégante dans Swift.

Il permet d’analyser JSON et de le mapper avec des classes de modèles personnalisées à l’aide d’un opérateur <-- :

```
identifier <-- json["id"]
name <-- json["name"]
stats <-- json["stats"]
```

Exemple:

Modèle rapide

```
struct Profile {
    var identifier = 0
    var name = ""
    var link: NSURL?
    var weekday: WeekDay = .Monday
    var stats = Stats()
    var phoneNumbers = [PhoneNumber]()
}
```

Fichier 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"
    }]
}
```

Cartographie

```
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"]
    }
}
```

Usage

```
let profile = Profile()
profile.deserialize(json)
```

Installation:

Carthage

```
github "s4cha/Arrow"
```

CacaoPods

```
pod 'Arrow'
use_frameworks!
```

Manuellement

Copiez et collez simplement Arrow.swift dans votre projet Xcode

https://github.com/s4cha/Arrow

Comme cadre

Téléchargez Arrow depuis le référentiel GitHub et créez la cible Framework sur le projet exemple. Puis lien contre ce cadre.

Analyse JSON simple dans des objets personnalisés

Même si les bibliothèques tierces sont bonnes, un moyen simple d'analyser le JSON est fourni par les protocoles. Vous pouvez imaginer que vous avez un objet Todo as

```
struct Todo {
    let comment: String
}
```

Chaque fois que vous recevez le JSON, vous pouvez gérer les NSData comme indiqué dans l'autre exemple utilisant l'objet NSJSONSerialization .

Après cela, en utilisant un protocole simple JSONDecodable

```
typealias JSONDictionary = [String:AnyObject]
protocol JSONDecodable {
    associatedtype Element
    static func from(json json: JSONDictionary) -> Element?
```

```
}
```

Et rendre votre structure Todo conforme à JSONDecodable fait le tour

```
extension Todo: JSONDecodable {
    static func from(json json: JSONDictionary) -> Todo? {
        guard let comment = json["comment"] as? String else { return nil }
        return Todo(comment: comment)
    }
}
```

Vous pouvez l'essayer avec ce code json:

```
{
    "todos": [
        {
            "comment" : "The todo comment"
        }
    ]
}
```

Lorsque vous l'avez obtenu à partir de l'API, vous pouvez le sérialiser comme les exemples précédents présentés dans une occurrence AnyObject . Après cela, vous pouvez vérifier si l'instance est une instance de JSONDictionary

```
guard let jsonDictionary = dictionary as? JSONDictionary else { return }
```

L'autre chose à vérifier, spécifique à ce cas car vous avez un tableau de Todo dans le JSON, est le dictionnaire todos

```
guard let todosDictionary = jsonDictionary["todos"] as? [JSONDictionary] else { return }
```

Maintenant que vous avez le tableau des dictionnaires, vous pouvez convertir chacun d'entre eux dans un objet Todo en utilisant flatMap (cela supprimera automatiquement les valeurs nil du tableau).

```
let todos: [Todo] = todosDictionary.flatMap { Todo.from(json: $0) }
```

JSON Parsing Swift 3

Voici le fichier JSON que nous utiliserons appelé 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?"
    }   ] }
```

Importez votre fichier JSON dans votre projet

Vous pouvez effectuer cette fonction simple pour imprimer votre fichier 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 vous voulez le placer dans une vue de table, je créerai d'abord un dictionnaire avec un NSObject.

Créez un nouveau fichier swift appelé ParsingObject et créez vos variables de chaîne.

Assurez-vous que le nom de la variable est le même que le fichier JSON

. Par exemple, dans notre projet, nous avons un name et une question . Dans notre nouveau fichier rapide, nous utiliserons

```
var name: String?
var question: String?
```

Initialisez le NSObject que nous avons renvoyé dans notre tableau var ViewController.swift = ParsingObject Ensuite, nous effectuerions la même méthode que précédemment avec une modification mineure.

```
      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)

             }

         }
```

Ensuite, nous le montrons dans notre tableview en faisant cela,

```
     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
         }
```

Lire Lecture et écriture JSON en ligne: https://riptutorial.com/fr/swift/topic/223/lecture-et-ecriture-json

Chapitre 38: Les extensions

Remarques

Vous pouvez en savoir plus sur les extensions dans The Swift Programming Language .

Examples

Variables et fonctions

Les extensions peuvent contenir des fonctions et des variables get / constantes. Ils sont dans le format

```
extension ExtensionOf {
    //new functions and get-variables
}
```

Pour référencer l'instance de l'objet étendu, self peut être utilisé, tout comme il pourrait être utilisé

Pour créer une extension de String qui ajoute une fonction .length() qui renvoie la longueur de la chaîne, par exemple

```
extension String {
    func length() -> Int {
        return self.characters.count
    }
}

"Hello, World!".length() // 13
```

Les extensions peuvent également contenir des variables get . Par exemple, ajouter une variable .length à la chaîne qui renvoie la longueur de la chaîne

```
extension String {
    var length: Int {
        get {
            return self.characters.count
        }
    }
}

"Hello, World!".length // 13
```

Initialiseurs dans les extensions

Les extensions peuvent contenir des initialiseurs de commodité. Par exemple, un initialiseur disponible pour Int qui accepte 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
```

Que sont les extensions?

Les extensions permettent d'étendre les fonctionnalités des types existants dans Swift. Les extensions peuvent ajouter des indices, des fonctions, des initialiseurs et des propriétés calculées. Ils peuvent également faire en sorte que les types soient conformes aux protocoles .

Supposons que vous souhaitiez pouvoir calculer la factorielle d'un Int . Vous pouvez ajouter une propriété calculée dans une extension:

```
extension Int {
    var factorial: Int {
        return (1..<self+1).reduce(1, combine: *)
    }
}
```

Vous pouvez ensuite accéder à la propriété comme si elle avait été incluse dans l'API Int d'origine.

```
let val1: Int = 10

val1.factorial  // returns 3628800
```

Extensions de protocole

Une fonctionnalité très utile de Swift 2.2 est la possibilité d’étendre les protocoles.

Cela fonctionne à peu près comme les classes abstraites quand il s'agit d'une fonctionnalité que vous voulez être disponible dans toutes les classes qui implémentent un protocole (sans devoir hériter d'une classe commune de base).

```
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
    }
}
```

Ceci est également possible en utilisant des génériques.

Restrictions

Il est possible d'écrire une méthode sur un type générique plus restrictif en utilisant where phrase.

```
extension Array where Element: StringLiteralConvertible {
  func toUpperCase() -> [String] {
      var result = [String]()
      for value in self {
          result.append(String(value).uppercaseString)
      }
      return result
    }
}
```

Exemple d'utilisation

```
let array = ["a","b","c"]
let resultado = array.toUpperCase()
```

Que sont les extensions et quand les utiliser

Les extensions ajoutent de nouvelles fonctionnalités à une classe, une structure, une énumération ou un type de protocole existant. Cela inclut la possibilité d'étendre les types pour lesquels vous n'avez pas accès au code source d'origine.

Les extensions dans Swift peuvent:

- Ajouter des propriétés calculées et des propriétés de type calculées
- Définir des méthodes d'instance et des méthodes de type
- Fournir de nouveaux initialiseurs
- Définir des indices
- Définir et utiliser de nouveaux types imbriqués
- Rendre un type existant conforme à un protocole

Quand utiliser les extensions Swift:

- Fonctionnalité supplémentaire à Swift
- Fonctionnalité supplémentaire pour UIKit / Foundation
- Fonctionnalité supplémentaire sans déranger le code des autres personnes
- Classes de décomposition en: Données / Fonctionnalité / Délégué

Quand ne pas utiliser:

- Étendre vos propres classes à partir d'un autre fichier

Exemple simple:

```
extension Bool {
    public mutating func toggle() -> Bool {
        self = !self
        return self
    }
}

var myBool: Bool = true
print(myBool.toggle()) // false
```

La source

Des indices

Les extensions peuvent ajouter de nouveaux indices à un type existant.

Cet exemple obtient le caractère à l'intérieur d'une chaîne en utilisant l'index donné:

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
```

Lire Les extensions en ligne: https://riptutorial.com/fr/swift/topic/324/les-extensions

Chapitre 39: Les fonctions

Examples

Utilisation de base

Les fonctions peuvent être déclarées sans paramètre ni valeur de retour. La seule information requise est un nom (hello dans ce cas).

```
func hello()
{
    print("Hello World")
}
```

Appelez une fonction sans paramètre en écrivant son nom suivi d'une paire de parenthèses vide.

```
hello()
//output: "Hello World"
```

Fonctions avec paramètres

Les fonctions peuvent prendre des paramètres pour que leurs fonctionnalités puissent être modifiées. Les paramètres sont donnés sous forme de liste séparée par des virgules, leurs types et noms étant définis.

```
func magicNumber(number1: Int)
{
    print("\(number1) Is the magic number")
}
```

Remarque: La syntaxe \(number1) est l' interpolation de base des chaînes et permet d'insérer le nombre entier dans la chaîne.

Les fonctions avec paramètres sont appelées en spécifiant la fonction par nom et en fournissant une valeur d'entrée du type utilisé dans la déclaration de fonction.

```
magicNumber(5)
//output: "5 Is the magic number
let example: Int = 10
magicNumber(example)
//output: "10 Is the magic number"
```

Toute valeur de type Int aurait pu être utilisée.

```
func magicNumber(number1: Int, number2: Int)
{
    print("\(number1 + number2) Is the magic number")
}
```

Lorsqu'une fonction utilise plusieurs paramètres, le nom du premier paramètre n'est pas requis pour le premier mais concerne les paramètres suivants.

```
let ten: Int = 10
let five: Int = 5
magicNumber(ten,number2: five)
```

```
//output: "15 Is the magic number"
```

Utilisez des noms de paramètres externes pour rendre les appels de fonctions plus lisibles.

```
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)
```

La définition de la valeur par défaut dans la déclaration de fonction vous permet d'appeler la fonction sans donner aucune valeur d'entrée.

```
func magicNumber(one number1: Int = 5, two number2: Int = 10)
{
    print("\(number1 + number2) Is the magic number")
}

magicNumber()
//output: "15 Is the magic number"
```

Valeurs de retour

Les fonctions peuvent renvoyer des valeurs en spécifiant le type après la liste des paramètres.

```
func findHypotenuse(a: Double, b: Double) -> Double
{
    return sqrt((a * a) + (b * b))
}

let c = findHypotenuse(3, b: 5)
//c = 5.830951894845301
```

Les fonctions peuvent également renvoyer plusieurs valeurs à l'aide de tuples.

```
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)
```

Erreurs de lancement

Si vous voulez une fonction pour pouvoir lancer des erreurs, vous devez ajouter le throws mot - clé après les parenthèses qui maintiennent les arguments:

```
func errorThrower()throws -> String {}
```

Lorsque vous souhaitez générer une erreur, utilisez le mot-clé throw :

```
func errorThrower()throws -> String {
  if true {
    return "True"
  } else {
    // Throwing an error
    throw Error.error
  }
}
```

Si vous souhaitez appeler une fonction pouvant générer une erreur, vous devez utiliser le mot-clé try dans un bloc do :

```
do {
  try errorThrower()
}
```

Pour plus d'informations sur les erreurs Swift: Erreurs

Les méthodes

Les méthodes d'instance sont des fonctions qui appartiennent à des instances d'un type dans Swift (une classe , une structure , une énumération ou un protocole). **Les méthodes de type** sont appelées sur un type lui-même.

Méthodes d'instance

Les méthodes d'instance sont définies avec une déclaration func dans la définition du type ou dans une extension .

```
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
}
```

La méthode d'instance increment() est appelée sur une instance de la classe Counter :

```
let counter = Counter()  // create an instance of Counter class
counter.increment()      // call the instance method on this instance
```

Type Méthodes

Les méthodes de type sont définies avec les mots-clés static func . (Pour les classes, class func définit une méthode de type pouvant être remplacée par des sous-classes.)

```
class SomeClass {
    class func someTypeMethod() {
        // type method implementation goes here
    }
}
```

```
SomeClass.someTypeMethod()  // type method is called on the SomeClass type itself
```

Paramètres Inout

Les fonctions peuvent modifier les paramètres qui leur sont transmis si elles sont marquées avec le mot-clé inout . Lors du passage d'un paramètre inout à une fonction, l'appelant doit ajouter un & à la variable transmise.

```
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".
```

Cela permet d'appliquer la sémantique de référence aux types qui auraient normalement une sémantique de valeur.

Syntaxe de fermeture de fuite

Lorsque le dernier paramètre d'une fonction est une fermeture

```
func loadData(id: String, completion:(result: String) -> ()) {
    // ...
    completion(result:"This is the result data")
}
```

la fonction peut être appelée à l'aide de la syntaxe de fermeture de suivi

```
loadData("123") { result in
    print(result)
}
```

Les opérateurs sont des fonctions

Des opérateurs tels que + , - , ?? sont une sorte de fonction nommée en utilisant des symboles plutôt que des lettres. Ils sont appelés différemment des fonctions:

- Préfixe: **-** x
- Infix: x **+** y
- Postfix: x **++**

Vous pouvez en savoir plus sur les opérateurs de base et les opérateurs avancés dans The Swift Programming Language.

Paramètres Variadic

Parfois, il n'est pas possible de répertorier le nombre de paramètres dont une fonction peut avoir besoin. Considérons une fonction de sum :

```
func sum(_ a: Int, _ b: Int) -> Int {
    return a + b
}
```

Cela fonctionne bien pour trouver la somme de deux nombres, mais pour trouver la somme de trois, il faudrait écrire une autre fonction:

```
func sum(_ a: Int, _ b: Int, _ c: Int) -> Int {
    return a + b + c
}
```

et un avec quatre paramètres en aurait besoin d'un autre, et ainsi de suite. Swift permet de définir une fonction avec un nombre variable de paramètres en utilisant une séquence de trois périodes: ... Par exemple,

```
func sum(_ numbers: Int...) -> Int {
    return numbers.reduce(0, combine: +)
}
```

Remarquez comment le paramètre de numbers , qui est variadique, est fusionné en un seul Array de type [Int] . Cela est vrai en général, les paramètres variadiques de type T... sont accessibles en tant que [T] .

Cette fonction peut maintenant être appelée comme ceci:

```
let a = sum(1, 2) // a == 3
let b = sum(3, 4, 5, 6, 7) // b == 25
```

Un paramètre variadique dans Swift ne doit pas nécessairement se trouver à la fin de la liste de paramètres, mais il ne peut y en avoir qu'un dans chaque signature de fonction.

Parfois, il est pratique de mettre une taille minimale sur le nombre de paramètres. Par exemple, cela n'a pas vraiment de sens de prendre la sum de aucune valeur. Un moyen simple d'imposer cela est de mettre des paramètres requis non variadiques, puis d'ajouter le paramètre variadic après. Pour être sûr que cette sum ne peut être appelée qu'avec au moins deux paramètres, nous pouvons écrire

```
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
```

Des indices

Les classes, les structures et les énumérations peuvent définir des indices, qui sont des raccourcis pour accéder aux éléments membres d'une collection, d'une liste ou d'une séquence.

Exemple

```
struct DaysOfWeek {

  var days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]

  subscript(index: Int) -> String {
    get {
      return days[index]
    }
    set {
      days[index] = newValue
    }
  }
```

```
}
```

Indice d'utilisation

```
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])
```

Options des indices:

Les indices peuvent prendre n'importe quel nombre de paramètres d'entrée, et ces paramètres d'entrée peuvent être de tout type. Les indices peuvent également renvoyer n'importe quel type. Les indices peuvent utiliser des paramètres variables et des paramètres variadiques, mais ne peuvent pas utiliser de paramètres d'entrée ou de sortie ni fournir des valeurs de paramètres par défaut.

Exemple:

```
struct Food {

  enum MealTime {
    case Breakfast, Lunch, Dinner
  }

  var meals: [MealTime: String] = [:]

  subscript (type: MealTime) -> String? {
    get {
      return meals[type]
    }
    set {
      meals[type] = newValue
    }
  }
}
```

Usage

```
var diet = Food()
diet[.Breakfast] = "Scrambled Eggs"
diet[.Lunch] = "Rice"

debugPrint("I had \(diet[.Breakfast]) for breakfast")
```

Fonctions avec fermetures

L'utilisation de fonctions qui intègrent et exécutent des fermetures peut être extrêmement utile pour envoyer un bloc de code à exécuter ailleurs. Nous pouvons commencer par permettre à notre fonction d'accepter une fermeture facultative qui, dans ce cas, renverra Void .

```
func closedFunc(block: (()->Void)? = nil) {
    print("Just beginning")

```

```
    if let block = block {
        block()
    }
}
```

Maintenant que notre fonction a été définie, appelons-la et transmettons du code:

```
closedFunc() { Void in
    print("Over already")
}
```

En utilisant une **fermeture finale** avec notre appel de fonction, nous pouvons transmettre du code (dans ce cas, print) pour être exécuté à un moment donné dans notre fonction closedFunc() .

Le journal devrait imprimer:

> Que commencer
>
> Déjà sur

Un cas d'utilisation plus spécifique pourrait inclure l'exécution de code entre deux classes:

```
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?()
    }
}
```

Le journal devrait imprimer:

> 1
>
> 2

Fonctions de passage et de retour

La fonction suivante renvoie une autre fonction en tant que résultat pouvant être affectée ultérieurement à une variable et appelée:

```
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)
```

Types de fonctions

Chaque fonction a son propre type de fonction, composé des types de paramètres et du type de retour de la fonction elle-même. Par exemple la fonction suivante:

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

a un type de fonction de:

```
(Int, Int) -> (Int)
```

Les types de fonction peuvent donc être utilisés comme types de paramètres ou comme types de retour pour les fonctions d'imbrication.

Lire Les fonctions en ligne: https://riptutorial.com/fr/swift/topic/432/les-fonctions

Chapitre 40: Méthode Swizzling

Remarques

Lorsque vous utilisez la méthode swizzling dans Swift, vos classes / méthodes doivent satisfaire à deux exigences:

- Votre classe doit étendre NSObject
- Les fonctions que vous souhaitez modifier doivent avoir l'attribut dynamic

Pour une explication complète de la nécessité de cette opération, consultez la rubrique Utilisation de Swift avec Cocoa et Objective-C :

> **Exiger un envoi dynamique**
>
> Bien que l'attribut @objc expose votre API Swift à l' @objc exécution Objective-C, il ne garantit pas la distribution dynamique d'une propriété, d'une méthode, d'un indice ou d'un initialiseur. *Le compilateur Swift peut toujours déstirualiser ou mettre en ligne l'accès des membres pour optimiser les performances de votre code, en contournant le runtime Objective-C* . Lorsque vous marquez une déclaration de membre avec le modificateur dynamic , l'accès à ce membre est toujours distribué dynamiquement. Les déclarations marquées avec le modificateur dynamic étant distribuées à l'aide de l' @objc exécution Objective-C, elles sont implicitement marquées avec l'attribut @objc .
>
> Exiger une répartition dynamique est rarement nécessaire. **Cependant, vous devez utiliser le modificateur dynamic lorsque vous savez que l'implémentation d'une API est remplacée lors de l'exécution** . Par exemple, vous pouvez utiliser la fonction method_exchangeImplementations dans l' method_exchangeImplementations exécution Objective-C pour remplacer l'implémentation d'une méthode pendant l'exécution d'une application. Si le compilateur Swift mettait en œuvre l'implémentation de la méthode ou y accédait, *la nouvelle implémentation ne serait pas utilisée* .

Liens

Référence à l'exécution d'Objective-C

Méthode Swizzling sur NSHipster

Examples

Extension de UIViewController et Swizzling viewDidLoad

En Objective-C, la méthode swizzling est le processus de modification de l'implémentation d'un sélecteur existant. Cela est possible grâce à la façon dont les sélecteurs sont mappés sur une table de distribution ou une table de pointeurs vers des fonctions ou des méthodes.

Les méthodes Pure Swift ne sont pas distribuées dynamiquement par le moteur d'exécution Objective-C, mais nous pouvons toujours tirer parti de ces astuces pour toute classe qui hérite de NSObject .

Ici, nous étendrons UIViewController et swizzle viewDidLoad pour ajouter de la journalisation personnalisée:

```
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()
    }
}
```

Bases de Swift Swizzling

methodOne() l'implémentation de methodOne() et de methodTwo() dans notre classe 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())
```

Bases de Swizzling - Objective-C

Exemple d'Objective-C de la méthode initWithFrame de initWithFrame:

```
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;
}
```

Lire Méthode Swizzling en ligne: https://riptutorial.com/fr/swift/topic/1436/methode-swizzling

Chapitre 41: Mise en cache sur l'espace disque

Introduction

Mise en cache de vidéos, d'images et d'audios à l'aide d' URLSession et de FileManager

Examples

Économie

```
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()
```

En train de lire

```
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
}
```

Lire Mise en cache sur l'espace disque en ligne:
https://riptutorial.com/fr/swift/topic/8902/mise-en-cache-sur-l-espace-disque

Chapitre 42: Nombres

Examples

Types de nombres et littéraux

Les types numériques intégrés à Swift sont:

- Taille Word (dépendant de l' architecture) signé `Int` et non signé `UInt` .
- Entiers signés de taille fixe `Int8` , `Int16` , `Int32` , `Int64` et entiers non signés `UInt8` , `UInt16` , `UInt32` , `UInt64` .
- `Float32 / Float` , `Float64 / Double` et `Float80` (x86 uniquement) de type à virgule flottante .

Littéraux

Le type d'un littéral numérique est déduit du contexte:

```
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
```

Underscores (`_`) peut être utilisé pour séparer les chiffres des littéraux numériques. Les zéros en tête sont ignorés.

Les littéraux à virgule flottante peuvent être spécifiés en utilisant des parties significand et exposant (*«significand»* **e** *«exponent»* pour la décimale; **0x** *«significand»* **p** *«exponent»* pour hexadécimal).

Entier la syntaxe littérale

```
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
```

Syntaxe littérale à virgule flottante

```
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 type numérique en un autre

```
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
```

Les initialiseurs entiers génèrent une **erreur d'exécution** si la valeur déborde ou est insuffisante:

```
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 conversion de flottant en entier **arrondit les valeurs à zéro** :

```
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 conversion entre nombres entiers peut être **avec perte** :

```
Int(Float(1_000_000_000_000_000_000))  // 999999984306749440
```

Convertir des nombres vers / depuis des chaînes

Utilisez les initialiseurs de chaîne pour convertir des nombres en chaînes:

```
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"
```

Ou utilisez l' interpolation de chaîne pour les cas simples:

```
let x = 42, y = 9001
"Between \(x) and \(y)"  // equivalent to "Between 42 and 9001"
```

Utilisez des initialiseurs de types numériques pour convertir des chaînes en nombres:

```
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)
```

Arrondi

rond

Arrondit la valeur au nombre entier le plus proche avec x.5 en arrondissant (mais notez que -x.5 arrondit à la baisse).

```
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
```

plafond

Arrondit n'importe quel nombre avec une valeur décimale au nombre entier supérieur suivant.

```
ceil(3.000) // 3
ceil(3.001) // 4
ceil(3.999) // 4

ceil(-3.000) // -3
ceil(-3.001) // -3
ceil(-3.999) // -3
```

sol

Arrondit tout nombre avec une valeur décimale au nombre entier inférieur.

```
floor(3.000) // 3
floor(3.001) // 3
floor(3.999) // 3

floor(-3.000) // -3
floor(-3.001) // -4
floor(-3.999) // -4
```

Int

Convertit un Double en Int , en supprimant toute valeur décimale.

```
Int(3.000) // 3
Int(3.001) // 3
Int(3.999) // 3

Int(-3.000) // -3
Int(-3.001) // -3
Int(-3.999) // -3
```

Remarques

- round , ceil et floor gérer à la fois l'architecture 64 et 32 bits.

Génération de nombres aléatoires

arc4random_uniform(someNumber: UInt32) -> UInt32

Cela vous donne des nombres entiers aléatoires compris entre 0 et someNumber - 1 .

La valeur maximale pour UInt32 est 4 294 967 295 (soit 2^32 - 1 UInt32).

Exemples:

- Flip de monnaie

  ```
  let flip = arc4random_uniform(2) // 0 or 1
  ```

- Jet de dés

  ```
  let roll = arc4random_uniform(6) + 1 // 1...6
  ```

- Journée au hasard en octobre

  ```
  let day = arc4random_uniform(31) + 1 // 1...31
  ```

- Année aléatoire dans les années 1990

  ```
  let year = 1990 + arc4random_uniform(10)
  ```

Forme générale:

```
let number = min + arc4random_uniform(max - min + 1)
```

où number , max et min sont UInt32 .

Remarques

- Il y a un léger biais modulo avec arc4random donc arc4random_uniform est préférable.
- Vous pouvez UInt32 une valeur UInt32 en Int mais UInt32 attention à ne pas sortir des limites.

Exponentiation

Dans Swift, nous pouvons **exposer** Double s avec la méthode pow() intégrée:

```
pow(BASE, EXPONENT)
```

Dans le code ci-dessous, la base (5) est mise à la puissance de l'exposant (2):

```
let number = pow(5.0, 2.0) // Equals 25
```

Lire Nombres en ligne: https://riptutorial.com/fr/swift/topic/454/nombres

Chapitre 43: NSRegularExpression dans Swift

Remarques

Caractères spéciaux

```
*?+[(){}^$|\./
```

Examples

Extension de la chaîne pour faire correspondre un modèle simple

```
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
```

Vous trouverez ci-dessous un autre exemple qui s'appuie sur ce qui précède pour faire quelque chose d'utile, qui ne peut être facilement fait par une autre méthode et se prête bien à une solution de regex.

```
// 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"))
```

Utilisation de base

Plusieurs considérations sont à prendre en compte lors de l'implémentation des expressions régulières dans 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"]
```

Afin d'obtenir une longueur de plage précise qui prend en charge tous les types de caractères, la chaîne d'entrée doit être convertie en une chaîne NSString.

Pour la sécurité, l'appariement par rapport à un modèle doit être inclus dans un bloc do catch pour gérer les défaillances

```
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 fonctionnalité d'expression régulière est souvent placée dans une extension ou une aide pour séparer les problèmes.

Remplacement des sous-couches

Les motifs peuvent être utilisés pour remplacer une partie d'une chaîne d'entrée.

L'exemple ci-dessous remplace le symbole du cent par le symbole du dollar.

```
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: "$")
} catch let error as NSError {
    print("Matching failed")
}
//correct$ = "$¥€£$¥€£$"
```

Caractères spéciaux

Pour faire correspondre des caractères spéciaux, vous devez utiliser une double barre oblique inverse \. becomes \\.

Les personnages que vous devrez échapper incluent

```
(){}[]/\+*$>.|^?
```

L'exemple ci-dessous comprend trois types de crochets

```
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 = ["(", "{", "["]
```

Validation

Les expressions régulières peuvent être utilisées pour valider les entrées en comptant le nombre de correspondances.

```
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 pour la validation du courrier

```
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)
}
```

ou vous pourriez utiliser l'extension de chaîne comme ceci:

```
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)
    }
}
```

Lire NSRegularExpression dans Swift en ligne:
https://riptutorial.com/fr/swift/topic/5763/nsregularexpression-dans-swift

Chapitre 44: Objets associés

Examples

Propriété, dans une extension de protocole, obtenue à l'aide d'un objet associé.

Dans Swift, les extensions de protocole ne peuvent pas avoir de vraies propriétés.

Cependant, dans la pratique, vous pouvez utiliser la technique "objet associé". Le résultat est presque exactement comme une "vraie" propriété.

Voici la technique exacte pour ajouter un "objet associé" à une extension de protocole:

Fondamentalement, vous utilisez les appels objective-c "objc_getAssociatedObject" et _set.

Les appels de base sont les suivants:

```
get {
   return objc_getAssociatedObject(self, & _Handle) as! YourType
   }
set {
   objc_setAssociatedObject(self, & _Handle, newValue, .OBJC_ASSOCIATION_RETAIN)
    }
```

Voici un exemple complet. Les deux points critiques sont:

1. Dans le protocole, vous devez utiliser ": class" pour éviter le problème de mutation.
2. Dans l'extension, vous devez utiliser "Where Self: UIViewController" (ou toute autre classe appropriée) pour indiquer le type de confirmation.

Donc, pour un exemple de propriété "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
}
```

Dans toute classe conforme, vous avez maintenant "ajouté" la propriété "p":

Vous pouvez utiliser "p" comme vous utiliseriez n'importe quelle propriété ordinaire de la classe conforme. Exemple:

```
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
    }
}
```

Remarque. Vous DEVEZ initialiser la pseudo-propriété.

Xcode ***ne*** vous **imposera** pas d'initialiser "p" dans la classe conforme.

Il est essentiel que vous initialisiez "p", peut-être dans viewDidLoad de la classe de confirmation.

Il convient de rappeler que p n'est en réalité qu'une ***propriété calculée*** . p n'est en réalité que deux fonctions, avec le sucre syntaxique. Il n'y a pas de p "variable" n'importe où: le compilateur "n'attribue pas de mémoire pour p" dans aucun sens. Pour cette raison, il est inutile de s'attendre à ce que Xcode impose "l'initialisation p".

En effet, pour parler plus précisément, vous devez vous rappeler d’utiliser "p pour la première fois, comme si vous l’initialisiez". (Encore une fois, ce serait très probablement dans votre code viewDidLoad.)

En ce qui concerne le getter en tant que tel.

Notez qu'il **va planter** si l'appel est appelé avant qu'une valeur pour "p" soit définie.

Pour éviter cela, considérez le code tel que:

```
    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
        }
```

Répéter. Xcode ***ne*** vous **imposera** pas l'initialisation p dans la classe conforme. Il est essentiel que vous initialisiez p, disons dans viewDidLoad de la classe conforme.

Rendre le code plus simple ...

Vous souhaiterez peut-être utiliser ces deux fonctions 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)
}
```

Notez qu'ils ne font rien, à part sauver la saisie et rendre le code plus lisible. (Ils sont essentiellement des macros ou des fonctions inline.)

Votre code devient alors:

```
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()
    }
}
```

(Dans l'exemple de _aoGet, P est initalisable: au lieu de P (), vous pouvez utiliser "", 0 ou toute valeur par défaut.)

Lire Objets associés en ligne: https://riptutorial.com/fr/swift/topic/1085/objets-associes

Chapitre 45: Opérateurs avancés

Examples

Opérateurs personnalisés

Swift prend en charge la création d'opérateurs personnalisés. Les nouveaux opérateurs sont déclarés au niveau global en utilisant le mot-clé operator .

La structure de l'opérateur est définie en trois parties: placement de l'opérande, priorité et associativité.

1. Les modificateurs de prefix , infix et postfix sont utilisés pour lancer une déclaration d'opérateur personnalisée. Les modificateurs de prefix et de postfix déclarent si l'opérateur doit être avant ou après, respectivement, la valeur sur laquelle il agit. Ces opérateurs sont urinaires, comme 8 et 3++ ** , car ils ne peuvent agir que sur une seule cible. L' infix déclare un opérateur binaire, qui agit sur les deux valeurs entre lesquelles il se trouve, telles que 2+3 .

2. Les opérateurs avec une **priorité** supérieure sont calculés en premier. La priorité de l'opérateur par défaut est juste supérieure à ? ... : (une valeur de 100 dans Swift 2.x). La priorité des opérateurs Swift standard peut être trouvée ici .

3. **L'associativité** définit l'ordre des opérations entre opérateurs de même priorité. Les opérateurs associatifs de gauche sont calculés de gauche à droite (ordre de lecture, comme la plupart des opérateurs), tandis que les opérateurs associatifs de droite calculent de droite à gauche.

3.0

À partir de Swift 3.0, on définirait la priorité et l’associativité dans un **groupe de priorité** au lieu de l’opérateur lui-même, de sorte que plusieurs opérateurs puissent facilement partager la même priorité sans faire référence aux nombres cryptés. La liste des groupes de priorité standard est indiquée ci - dessous .

Les opérateurs renvoient des valeurs en fonction du code de calcul. Ce code agit comme une fonction normale, avec des paramètres spécifiant le type d'entrée et le mot-clé de return spécifiant la valeur calculée que l'opérateur retourne.

Voici la définition d'un opérateur exponentiel simple, car Swift standard n'a pas d'opérateur exponentiel.

```
import Foundation

infix operator ** { associativity left precedence 170 }

func ** (num: Double, power: Double) -> Double{
    return pow(num, power)
}
```

L' infix indique que l'opérateur ** travaille entre deux valeurs, telles que 9**2 . Comme la fonction a laissé une associativité, 3**3**2 est calculé comme suit: (3**3)**2 . La priorité de 170 est supérieure à toutes les opérations Swift standard, ce qui signifie que 3+2**4 calcule à 19 , malgré l’associativité de gauche de ** .

3.0

```
import Foundation

infix operator **: BitwiseShiftPrecedence
```

```
func ** (num: Double, power: Double) -> Double {
    return pow(num, power)
}
```

Au lieu de spécifier explicitement la préséance et l'associativité, sur Swift 3.0, nous pourrions utiliser le groupe de priorité intégré BitwiseShiftPrecedence qui donne les valeurs correctes (identiques à << , >>).

**: L'incrémentation et la décrémentation sont obsolètes et seront supprimées dans Swift 3.

Surcharge + pour les dictionnaires

Comme il n'existe actuellement aucun moyen simple de combiner des dictionnaires dans Swift, il peut être utile de surcharger les opérateurs + et += pour ajouter cette fonctionnalité à l'aide de génériques .

```
// 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

À partir de Swift 3, inout doit être placé avant le type d'argument.

```
func +=<K, V>(lhs: inout [K : V], rhs: [K : V]) { ... }
```

Exemple d'utilisation:

```
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"]
```

Opérateurs de communication

Ajoutons un opérateur personnalisé pour multiplier un CGSize

```
func *(lhs: CGFloat, rhs: CGSize) -> CGSize{
    let height = lhs*rhs.height
    let width = lhs*rhs.width
    return CGSize(width: width, height: height)
}
```

Maintenant cela fonctionne

```
let sizeA = CGSize(height:100, width:200)
let sizeB = 1.1 * sizeA         //=> (height: 110, width: 220)
```

Mais si nous essayons de faire l'opération en sens inverse, nous obtenons une erreur

```
let sizeC = sizeB * 20          // ERROR
```

Mais c'est assez simple d'ajouter:

```
func *(lhs: CGSize, rhs: CGFloat) -> CGSize{
    return rhs*lhs
}
```

Maintenant, l'opérateur est commutatif.

```
let sizeA = CGSize(height:100, width:200)
let sizeB = sizeA * 1.1              //=> (height: 110, width: 220)
```

Opérateurs sur les bits

Les opérateurs Swift Bitwise vous permettent d'effectuer des opérations sur la forme binaire des nombres. Vous pouvez spécifier un littéral binaire en préfixant le nombre avec 0b . Par exemple, 0b110 équivaut au nombre binaire 110 (nombre décimal 6). Chaque 1 ou 0 est un peu dans le nombre.

Bitwise NOT ~ :

```
var number: UInt8 = 0b01101100
let newNumber = ~number
// newNumber is equal to 0b01101100
```

Ici, chaque bit se transforme en son contraire. Déclarer explicitement le nombre UInt8 garantit que le nombre est positif (pour que nous n'ayons pas à traiter de négatifs dans l'exemple) et qu'il ne s'agisse que de 8 bits. Si 0b01101100 était un UInt plus grand, il y aurait des 0 qui seraient convertis en 1 et deviendraient significatifs lors de l'inversion:

```
var number: UInt16 = 0b01101100
// number equals 0b0000000001101100
// the 0s are not significant
let newNumber = ~number
// newNumber equals 0b1111111110010011
// the 1s are now significant
```

- 0 à> 1
- 1 -> 0

Bitwise AND & :

```
var number = 0b0110
let newNumber = number & 0b1010
// newNumber is equal to 0b0010
```

Ici, un bit donné sera 1 si et seulement si les nombres binaires des deux côtés de l'opérateur &

contenaient un 1 à cet endroit du bit.

- 0 & 0 -> 0
- 0 & 1 -> 0
- 1 & 1 -> 1

Bit à bit OU | :

```
var number = 0b0110
let newNumber = number | 0b1000
// newNumber is equal to 0b1110
```

Ici, un bit donné sera 1 si et seulement si le nombre binaire sur au moins un côté du | l'opérateur contenait un 1 à cet endroit du bit.

- 0 | 0 -> 0
- 0 | 1 -> 1
- 1 | 1 -> 1

Bit à bit XOR (OU exclusif) ^ :

```
var number = 0b0110
let newNumber = number ^ 0b1010
// newNumber is equal to 0b1100
```

Ici, un bit donné sera 1 si et seulement si les bits dans cette position des deux opérandes sont différents.

- 0 ^ 0 -> 0
- 0 ^ 1 -> 1
- 1 ^ 1 -> 0

Pour toutes les opérations binaires, l'ordre des opérandes ne fait aucune différence sur le résultat.

Opérateurs de débordement

Le dépassement de capacité se rapporte à ce qui se produit lorsqu'une opération aboutit à un nombre supérieur ou inférieur à la quantité de bits spécifiée pour ce numéro.

En raison de la façon dont fonctionne l'arithmétique binaire, après qu'un nombre soit devenu trop grand pour ses bits, le nombre déborde jusqu'au plus petit nombre possible (pour la taille du bit) et continue ensuite à compter. De la même manière, lorsqu'un nombre devient trop petit, il débouche sur le plus grand nombre possible (pour sa taille en bits) et continue à décompter à partir de là.

Étant donné que ce comportement n'est pas souvent souhaité et peut entraîner de graves problèmes de sécurité, les opérateurs arithmétiques Swift + , - et * émet des erreurs lorsqu'une opération provoque un débordement ou un débordement. Pour autoriser explicitement le dépassement et le sous-dépassement, utilisez plutôt &+ , &- et &* .

```
var almostTooLarge = Int.max
almostTooLarge + 1 // not allowed
almostTooLarge &+ 1 // allowed, but result will be the value of Int.min
```

Préséance des opérateurs Swift standard

Les opérateurs liés plus fortement (priorité supérieure) sont listés en premier.

Les opérateurs	Groupe de préséance (≥3.0)	Priorité	Associativité
.		∞	la gauche
? , ! , ++ , -- , [] , () , {}	(postfix)		
! , ~ , + , - , ++ , --	(préfixe)		
~> (swift ≤2.3)		255	la gauche
<< , >>	BitwiseShiftPrecedence	160	aucun
* , / , % , & , &*	MultiplicationPrecedence	150	la gauche
+ , - , | , ^ , &+ , &-	AdditionPrecedence	140	la gauche
... , ..<	RangeFormationPrecedence	135	aucun
is , as , as? , as!	CastingPrecedence	132	la gauche
??	NilCoalescingPrecedence	131	droite
< , <= , > , >= , == != , === !== , ~=	ComparaisonPrécédence	130	aucun
&&	LogicalConjunctionPrecedence	120	la gauche
||	LogicalDisjunctionPrecedence	110	la gauche
	DefaultPrecedence *		aucun
? ... :	TernaryPrecedence	100	droite
= , += , -= , *= , /= , %= , <<= , >>= , &= , |= , ^=	AssignmentPrecedence	90	droit, cession
->	FunctionArrowPrecedence		droite

3.0

- Le groupe de priorité DefaultPrecedence est supérieur à TernaryPrecedence , mais il n'est pas ordonné avec le reste des opérateurs. Outre ce groupe, les autres priorités sont linéaires.

- Cette table se trouve également sur la référence API d'Apple
- La définition réelle des groupes de priorité peut être trouvée dans le code source sur GitHub.

Lire Opérateurs avancés en ligne: https://riptutorial.com/fr/swift/topic/1048/operateurs-avances

Chapitre 46: Options

Introduction

«Une valeur facultative contient une valeur ou contient une valeur nulle pour indiquer qu'une valeur est manquante»

Extrait de: Apple Inc. «Le langage de programmation rapide (Swift 3.1 Edition)». IBooks. https://itun.es/us/k5SW7.l

Les cas d'utilisation optionnels de base incluent: pour une constante (let), utiliser une option dans une boucle (if-let), dérouler en toute sécurité une valeur optionnelle dans une méthode (guard-let), et faire partie des boucles de commutation (casse), par défaut à une valeur nulle, en utilisant l'opérateur coalesce (??)

Syntaxe

- var optionalName: optionalType? // déclare un type optionnel, par défaut à nil
- var optionalName: optionalType? = valeur // déclare une option avec une valeur
- var optionalName: optionalType! // déclare une option implicitement déballée
- optionnel! // force à déplier une option

Remarques

Pour plus d'informations sur les options, voir Le langage de programmation Swift .

Examples

Types d'option

Les options sont un type enum générique qui agit comme un wrapper. Ce wrapper permet à une variable d'avoir l'un des deux états suivants: la valeur du type défini par l'utilisateur ou nil , qui représente l'absence de valeur.

Cette capacité est particulièrement importante pour Swift car l'un des objectifs de conception du langage est de bien fonctionner avec les frameworks Apple. Beaucoup (la plupart) des frameworks Apple utilisent nil raison de sa facilité d'utilisation et de son importance pour les modèles de programmation et la conception d'API dans Objective-C.

Dans Swift, pour qu'une variable ait une valeur nil , elle doit être facultative. Les options peuvent être créées en ajoutant soit a ! ou un ? au type de variable. Par exemple, pour rendre un Int optionnel, vous pouvez utiliser

```
var numberOne: Int! = nil
var numberTwo: Int? = nil
```

? Les options doivent être explicitement déballées et doivent être utilisées si vous n'êtes pas certain si la variable aura une valeur lorsque vous y accéderez. Par exemple, lorsque vous transformez une chaîne en Int , le résultat est un Int? optionnel Int? , car nil sera retourné si la chaîne n'est pas un nombre valide

```
let str1 = "42"
let num1: Int? = Int(str1) // 42

let str2 = "Hello, World!"
let num2: Int? = Int(str2) // nil
```

! optionals sont déballés automatiquement et *ne* doivent être utilisés lorsque vous êtes *certain*

que la variable aura une valeur lorsque vous y accédez. Par exemple, un UIButton! mondial UIButton! variable initialisée dans 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)
}
```

Déballer une option

Pour accéder à la valeur d'une option, elle doit être déballée.

Vous pouvez *déballer conditionnellement* une option en utilisant une liaison facultative et *forcer* un *dépliage* facultatif en utilisant le ! opérateur.

Déballage conditionnel demande efficacement "Cette variable a-t-elle une valeur?" tandis que le dépliage forcé indique "Cette variable a une valeur!".

Si vous forcez de déplier une variable qui est nil , votre programme lancera un *néant trouvé de manière inattendue tout en déroulant une* exception *facultative* et un crash, vous devrez donc examiner attentivement si vous utilisez ! est approprié.

```
var text: String? = nil
var unwrapped: String = text! //crashes with "unexpectedly found nil while unwrapping an
Optional value"
```

Pour un défilement sécurisé, vous pouvez utiliser une instruction if-let , qui ne lancera pas d'exception ni de blocage si la valeur nil est 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")
}
```

Ou une déclaration de garde :

```
var number: Int?
guard let unwrappedNumber = number else {
    return
}
print("number: \(unwrappedNumber)")
```

Notez que la portée de la variable unwrappedNumber trouve dans l'instruction if-let et à l'extérieur du bloc de guard .

Vous pouvez enchaîner le dépliage de plusieurs options, cela est principalement utile dans les cas où votre code nécessite plus de variables pour s'exécuter correctement:

```
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.
}
```

Notez que toutes les variables doivent être déroulées pour réussir le test, sinon vous n'auriez aucun moyen de déterminer quelles variables ont été déballées et lesquelles ne l'ont pas été.

Vous pouvez enchaîner les instructions conditionnelles en utilisant vos options après leur déballage. Cela signifie pas d'instructions imbriquées if - else!

```
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!")
}
```

Opérateur de coalescence néant

Vous pouvez utiliser l' *opérateur de coalescence nil* pour déballer une valeur si elle est non-nulle, sinon indiquez une valeur différente:

```
func fallbackIfNil(str: String?) -> String {
    return str ?? "Fallback String"
}
print(fallbackIfNil("Hi")) // Prints "Hi"
print(fallbackIfNil(nil)) // Prints "Fallback String"
```

Cet opérateur est capable de court-circuiter , ce qui signifie que si l'opérande gauche est non nul, le bon opérande ne sera pas évalué:

```
func someExpensiveComputation() -> String { ... }

var foo : String? = "a string"
let str = foo ?? someExpensiveComputation()
```

Dans cet exemple, comme foo est non-nil, someExpensiveComputation() ne sera pas appelé.

Vous pouvez également enchaîner plusieurs déclarations de coalescence nulle:

```
var foo : String?
var bar : String?

let baz = foo ?? bar ?? "fallback string"
```

Dans cet exemple, baz verra attribuer la valeur non foo de foo si elle est non-nulle, sinon la valeur non-emballée de bar lui sera attribuée si elle est non-nulle, sinon la valeur de repli lui sera attribuée.

Chaînage en option

Vous pouvez utiliser le chaînage facultatif pour appeler une méthode , accéder à une propriété ou à un indice en option. Cela se fait en plaçant un ? entre la variable facultative donnée et le membre donné (méthode, propriété ou indice).

```
struct Foo {
    func doSomething() {
        print("Hello World!")
    }
}

var foo : Foo? = Foo()

foo?.doSomething() // prints "Hello World!" as foo is non-nil
```

Si foo contient une valeur, doSomething() sera appelé dessus. Si foo est nil , alors rien ne se passera - le code échouera simplement en silence et continuera à s'exécuter.

```
var foo : Foo? = nil

foo?.doSomething() // will not be called as foo is nil
```

(Ceci est un comportement similaire à l'envoi de messages à nil dans Objective-C)

La raison pour laquelle le chaînage facultatif est nommé en tant que tel est que «optionality» sera propagé à travers les membres que vous appelez / accédez. Cela signifie que les valeurs de retour des membres utilisés avec un chaînage facultatif seront facultatives, qu'elles soient ou non facultatives.

```
struct Foo {
    var bar : Int
    func doSomething() { ... }
}

let foo : Foo? = Foo(bar: 5)
print(foo?.bar) // Optional(5)
```

Ici foo?.bar renvoie un Int? Bien que la bar soit pas facultative, foo lui-même est facultatif.

Comme optionalité se propage, les méthodes de retour Void renverront Void? lorsqu'il est appelé avec chaînage facultatif. Cela peut être utile pour déterminer si la méthode a été appelée ou non (et donc si l'option a une valeur).

```
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")
}
```

Ici, nous comparons le Void? retourner la valeur avec nil afin de déterminer si la méthode a été appelée (et donc si foo est non-nil).

Vue d'ensemble - Pourquoi les options?

Souvent, lors de la programmation, il est nécessaire de faire une distinction entre une variable qui a une valeur et une autre qui ne l'a pas. Pour les types de référence, tels que les pointeurs C, une valeur spéciale telle que null peut être utilisée pour indiquer que la variable n'a aucune valeur. Pour les types intrinsèques, comme un entier, cela est plus difficile. Une valeur nominée, telle que -1, peut être utilisée, mais cela dépend de l'interprétation de la valeur. Il élimine également cette valeur "spéciale" de l'utilisation normale.

Pour résoudre ce problème, Swift permet à toute variable d'être déclarée comme facultative. Ceci

est indiqué par l'utilisation d'un? ou ! après le type (voir Types de optionnels)

Par exemple,

```
var possiblyInt: Int?
```

déclare une variable pouvant ou non contenir une valeur entière.

La valeur spéciale nil indique qu'aucune valeur n'est actuellement affectée à cette variable.

```
possiblyInt = 5      // PossiblyInt is now 5
possiblyInt = nil    // PossiblyInt is now unassigned
```

nil peut également être utilisé pour tester une valeur assignée:

```
if possiblyInt != nil {
    print("possiblyInt has the value \(possiblyInt!)")
}
```

Notez l'utilisation de ! dans l'instruction print pour *dérouler* la valeur facultative.

À titre d'exemple d'utilisation commune des options, considérez une fonction qui renvoie un entier à partir d'une chaîne contenant des chiffres; Il est possible que la chaîne contienne des caractères non numériques ou même vide.

Comment une fonction qui retourne un simple Int indiquer un échec? Il ne peut pas le faire en renvoyant une valeur spécifique car cela empêcherait cette valeur d'être analysée à partir de la chaîne.

```
var someInt
someInt = parseInt("not an integer") // How would this function indicate failure?
```

Dans Swift, cependant, cette fonction peut simplement renvoyer une *option* Int. Alors, l'échec est indiqué par la valeur de retour 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")
}
```

Lire Options en ligne: https://riptutorial.com/fr/swift/topic/247/options

Chapitre 47: OptionSet

Examples

Protocole OptionSet

OptionSetType est un protocole conçu pour représenter les types de masque de bits où les bits individuels représentent les membres de l'ensemble. Un ensemble de fonctions logiques et / ou de fonctions applique la syntaxe appropriée:

```
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
```

Lire OptionSet en ligne: https://riptutorial.com/fr/swift/topic/1242/optionset

Chapitre 48: Performance

Examples

Performance d'allocation

Dans Swift, la gestion de la mémoire se fait automatiquement à l'aide du comptage automatique des références. (Voir Gestion de la mémoire) L'allocation est le processus de réservation d'un emplacement en mémoire pour un objet, et dans une compréhension rapide, ces performances nécessitent une certaine compréhension du **tas** et de la **pile** . Le tas est un emplacement de mémoire où la plupart des objets sont placés et vous pouvez le considérer comme un entrepôt de stockage. La pile, en revanche, est une pile d'appels de fonctions ayant conduit à l'exécution en cours. (Par conséquent, une trace de pile est une sorte d'impression des fonctions de la pile d'appels.)

L'allocation et la désallocation de la pile est une opération très efficace, mais en comparaison, l'allocation de tas est coûteuse. Lors de la conception pour la performance, vous devez garder cela à l'esprit.

Des classes:

```
class MyClass {

    let myProperty: String

}
```

Les classes dans Swift sont des types de référence et par conséquent plusieurs choses se produisent. Tout d'abord, l'objet réel sera alloué au tas. Toutes les références à cet objet doivent ensuite être ajoutées à la pile. Cela fait des classes un objet plus coûteux à attribuer.

Structs:

```
struct MyStruct {

    let myProperty: Int

}
```

Les structures étant des types de valeur et donc copiées lorsqu elles sont transmises, elles sont allouées sur la pile. Cela rend les structures plus efficaces que les classes, cependant, si vous avez besoin d'une notion d'identité et / ou de sémantique de référence, une structure ne peut pas vous fournir ces choses.

Avertissement sur les structures avec des chaînes et des propriétés qui sont des classes

Bien que les structs soient généralement plus sympas que les classes, vous devez faire attention aux structs avec des propriétés qui sont des classes:

```
struct MyStruct {

    let myProperty: MyClass

}
```

Ici, en raison du comptage de références et d'autres facteurs, la performance est maintenant plus similaire à une classe. De plus, si plusieurs propriétés de la structure sont une classe, l'impact sur les performances peut être encore plus négatif que si la structure était plutôt une

classe.

De plus, bien que les chaînes soient des structures, elles stockent en interne leurs caractères sur le tas, et sont donc plus chères que la plupart des structures.

Lire Performance en ligne: https://riptutorial.com/fr/swift/topic/4067/performance

Chapitre 49: Premiers pas avec la programmation orientée protocole

Remarques

Pour plus d'informations sur ce sujet, reportez-vous à la section consacrée à la programmation orientée protocole WWDC 2015 de Swift .

Il existe également un excellent guide écrit sur le même sujet: Introduction à la programmation orientée protocole dans Swift 2 .

Examples

Utilisation de la programmation orientée protocole pour les tests unitaires

La programmation orientée protocole est un outil utile pour écrire facilement de meilleurs tests unitaires pour notre code.

Disons que nous voulons tester un UIViewController qui repose sur une classe ViewModel.

Les étapes nécessaires sur le code de production sont les suivantes:

1. Définissez un protocole qui expose l'interface publique de la classe ViewModel, avec toutes les propriétés et méthodes requises par UIViewController.
2. Implémentez la vraie classe ViewModel, conforme à ce protocole.
3. Utilisez une technique d'injection de dépendance pour permettre au contrôleur de vue d'utiliser l'implémentation souhaitée, en le transmettant en tant que protocole et non en tant qu'instance concrète.

```
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")
```

Ensuite, sur test unitaire:

1. Implémenter un modèle ViewModel simulé conforme au même protocole
2. Transmettez-le à UIViewController sous test en utilisant l'injection de dépendance, au lieu de l'instance réelle.
3. Tester!

```
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)
    }
}
```

Utilisation de protocoles comme types de première classe

La programmation orientée protocole peut être utilisée comme modèle de conception Swift.

Différents types peuvent se conformer au même protocole, les types de valeur peuvent même se conformer à plusieurs protocoles et même fournir une implémentation de méthode par défaut.

Initialement, des protocoles peuvent être définis pour représenter des propriétés et / ou des méthodes couramment utilisées avec des types spécifiques ou génériques.

```
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)
}
```

Une implémentation par défaut de la méthode get peut être créée, mais si vous le souhaitez, les types conformes peuvent remplacer l'implémentation.

```
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 type de valeur conforme au protocole ItemData, ce type de valeur peut également se conformer à d'autres protocoles.

```
struct Item: ItemData {

    let title: String
    let description: String
```

```
    let thumbnailURL: NSURL
    let created: NSDate
    let updated: NSDate

}
```

Ici, la structure d'élément est étendue pour se conformer à un élément d'affichage.

```
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 exemple de site d'appel pour l'utilisation de la méthode get statique.

```
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
    })

}
```

Différents cas d'utilisation nécessiteront des implémentations différentes. L'idée principale ici est de montrer la conformité de différents types lorsque le protocole est le point principal de la conception. Dans cet exemple, les données de l'API sont peut-être sauvegardées sous condition dans une entité Core Data.

```
// 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
}
```

Ici, la classe sauvegardée Core Data peut également être conforme au protocole 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)
        }

    }
}
```

Lire Premiers pas avec la programmation orientée protocole en ligne: https://riptutorial.com/fr/swift/topic/2502/premiers-pas-avec-la-programmation-orientee-protocole

Chapitre 50: Programmation fonctionnelle dans Swift

Examples

Extraire une liste de noms d'une liste de personne (s)

Étant donné une Person struct

```
struct Person {
    let name: String
    let birthYear: Int?
}
```

et un tableau 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)
]
```

nous pouvons récupérer un tableau de String contenant la propriété name de chaque personne.

```
let names = persons.map { $0.name }
// ["Walter White", "Jesse Pinkman", "Skyler White", "Saul Goodman"]
```

Traversée

```
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))
}
```

En saillie

Appliquer une fonction à une collection / flux et créer une nouvelle collection / flux s'appelle une **projection** .

```
/// 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)
```

Filtration

La création d'un flux en sélectionnant les éléments d'un flux qui passe une certaine condition est appelée **filtrage**

```
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)
```

Utiliser un filtre avec Structs

Vous souhaiterez peut-être fréquemment filtrer les structures et autres types de données complexes. La recherche d'un tableau de structures pour des entrées contenant une valeur particulière est une tâche très courante et facilement réalisée dans Swift à l'aide de fonctions de programmation fonctionnelles. De plus, le code est extrêmement succinct.

```
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"
```

Lire Programmation fonctionnelle dans Swift en ligne:
https://riptutorial.com/fr/swift/topic/2948/programmation-fonctionnelle-dans-swift

Chapitre 51: Protocoles

Introduction

Les protocoles sont un moyen de spécifier comment utiliser un objet. Ils décrivent un ensemble de propriétés et de méthodes qu'une classe, une structure ou un enum devrait fournir, bien que les protocoles ne posent aucune restriction à l'implémentation.

Remarques

Un protocole Swift est un ensemble d'exigences que les types conformes doivent implémenter. Le protocole peut ensuite être utilisé dans la plupart des endroits où un type est attendu, par exemple des tableaux et des exigences génériques.

Les membres du protocole partagent toujours le même qualificateur d'accès que le protocole complet et ne peuvent pas être spécifiés séparément. Bien qu'un protocole puisse restreindre l'accès avec les exigences de getter ou de setter, comme dans les exemples ci-dessus.

Pour plus d'informations sur les protocoles, voir Le langage de programmation Swift .

Les protocoles Objective-C sont similaires aux protocoles Swift.

Les protocoles sont également comparables aux interfaces Java .

Examples

Principes de base du protocole

À propos des protocoles

Un protocole spécifie les initialiseurs, les propriétés, les fonctions, les indices et les types associés requis pour un type d'objet Swift (classe, struct ou enum) conforme au protocole. Dans certaines langues, des idées similaires pour les spécifications d'exigences des objets suivants sont appelées «interfaces».

Un protocole déclaré et défini est un Type, en soi, avec une signature de ses exigences déclarées, quelque peu similaire à la manière dont les Fonctions Swift sont un Type basé sur leur signature de paramètres et de retours.

Les spécifications du protocole Swift peuvent être facultatives, explicitement requises et / ou implémentées par défaut via une fonction appelée Extensions de protocole. Un type d'objet Swift (classe, struct ou enum) souhaitant se conformer à un protocole étoffé avec les extensions pour toutes ses exigences spécifiées doit seulement indiquer son désir de se conformer à la conformité totale. La fonctionnalité d'implémentation par défaut des extensions de protocole peut suffire à remplir toutes les obligations de conformité à un protocole.

Les protocoles peuvent être hérités par d'autres protocoles. Ceci, conjointement avec les extensions de protocole, signifie que les protocoles peuvent et doivent être considérés comme une caractéristique importante de Swift.

Les protocoles et les extensions sont importants pour réaliser les objectifs et les approches plus larges de Swift en matière de flexibilité de conception de programme et de processus de développement. Le principal objectif déclaré de la fonctionnalité Protocole et extension de Swift est la facilitation de la conception de la composition dans l'architecture et le développement des programmes. Ceci est appelé programmation orientée protocole. Les anciens croustillants considèrent cela comme supérieur à la conception de la POO.

Les protocoles définissent des interfaces pouvant être implémentées par toute structure , classe ou énumération :

```
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
}
```

Les propriétés définies dans les protocoles doivent être annotées sous la forme { get } ou { get set } . { get } signifie que la propriété doit être mémorable et peut donc être implémentée comme *n'importe quel* type de propriété. { get set } signifie que la propriété doit être paramétrable et inoubliable.

Une structure, une classe ou un enum peut **être conforme à** un protocole:

```
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 protocole peut également définir une **implémentation par défaut** pour l'une de ses exigences via une extension :

```
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
    }
}
```

Un protocole peut être **utilisé en tant que type** , à condition qu'il ne soit pas associatedtype exigences de type :

```
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]
```

Vous pouvez également définir un type abstrait conforme à **plusieurs** protocoles:

3.0

Avec Swift 3 ou supérieur, cela se fait en séparant la liste des protocoles avec une esperluette (&):

```
func doStuff(object: MyProtocol & AnotherProtocol) {
    // ...
}

let items : [MyProtocol & AnotherProtocol] = [MyStruct(), MyClass(), MyEnum.caseA]
```

3.0

Les anciennes versions ont un protocol<...> syntaxe protocol<...> où les protocoles sont une liste séparée par des virgules entre les crochets angulaires <> .

```
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]
```

Les types existants peuvent être **étendus** pour se conformer à un protocole:

```
extension String : MyProtocol {
    // Implement any requirements which String doesn't already satisfy
}
```

Exigences de type associé

Les protocoles peuvent définir des **exigences de type associés** à l' aide du associatedtype mot - clé:

```
protocol Container {
    associatedtype Element
    var count: Int { get }
    subscript(index: Int) -> Element { get set }
}
```

Les protocoles avec des exigences de type associées ne **peuvent être utilisés que comme contraintes génériques** :

```
// 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 type conforme au protocole peut satisfaire implicitement à une exigence de type associatedtype , en fournissant un type donné où le protocole s'attend à ce que le type associatedtype apparaisse:

```
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")
```

(Notez que pour ajouter de la clarté à cet exemple, le type d'espace réservé générique est nommé T - un nom plus approprié serait l' Element , qui ombre du protocole d' associatedtype Element . Le compilateur déduira encore que l'espace réservé générique Element est utilisé pour satisfaire les associatedtype Element requis.)

Un type associatedtype peut également être satisfait explicitement par l'utilisation d'un typealias :

```
struct ContainerOfOne<T>: Container {

    typealias Element = T
    subscript(index: Int) -> Element { ... }

    // ...
}
```

La même chose vaut pour les extensions:

```
// 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 le type conforme répond déjà à l'exigence, aucune implémentation n'est requise:

```
extension Array: Container {}  // Array satisfies all requirements, including Element
```

Modèle de délégué

Un *délégué* est un modèle de conception commun utilisé dans les frameworks Cocoa et CocoaTouch, où une classe délègue la responsabilité de l'implémentation de certaines fonctionnalités à une autre. Cela suit un principe de séparation des préoccupations, dans lequel la classe d'infrastructure implémente des fonctionnalités génériques alors qu'une instance de délégué distinct implémente le cas d'utilisation spécifique.

Une autre manière d’examiner le modèle des délégués est la communication par objet. Objects doivent souvent se parler et, pour ce faire, un objet doit être conforme à un protocol pour devenir un délégué d'un autre objet. Une fois cette configuration effectuée, l’autre objet parle à ses délégués lorsque des choses intéressantes se produisent.

Par exemple, une vue dans l'interface utilisateur pour afficher une liste de données ne devrait être responsable que de la logique d'affichage des données et non pas de la détermination des données à afficher.

Plongeons dans un exemple plus concret. si vous avez deux classes, un parent et un enfant:

```
class Parent { }
class Child { }
```

Et vous voulez informer le parent d'un changement de l'enfant.

Dans Swift, les délégués sont implémentés en utilisant une déclaration de protocol et nous allons donc déclarer un protocol que le delegate implémentera. Ici, delegate est l'objet parent .

```
protocol ChildDelegate: class {
    func childDidSomething()
}
```

L'enfant doit déclarer une propriété pour stocker la référence au délégué:

```
class Child {
    weak var delegate: ChildDelegate?
}
```

Notez la variable delegate est en option et le protocole ChildDelegate est marqué à seulement mis en œuvre par type de classe (sans que cela le delegate variable ne peut être déclarée comme une weak référence évitant tout cycle retenir. Cela signifie que si le delegate variable n'est plus référencé ailleurs, il sera publié). La classe parente enregistre uniquement le délégué lorsque cela est nécessaire et disponible.

Aussi, pour marquer notre délégué comme weak nous devons contraindre notre protocole ChildDelegate à référencer les types en ajoutant class mot class clé de class dans la déclaration de protocole.

Dans cet exemple, lorsque l'enfant fait quelque chose et doit notifier son parent, l'enfant appelle:

```
delegate?.childDidSomething()
```

Si le délégué a été défini, le délégué sera informé que l'enfant a fait quelque chose.

La classe parente devra étendre le protocole ChildDelegate pour pouvoir répondre à ses actions. Cela peut être fait directement sur la classe parente:

```
class Parent: ChildDelegate {
    ...

    func childDidSomething() {
        print("Yay!")
    }
}
```

Ou en utilisant une extension:

```
extension Parent: ChildDelegate {
    func childDidSomething() {
        print("Yay!")
    }
}
```

Le parent doit également dire à l'enfant que c'est le délégué de l'enfant:

```
// In the parent
let child = Child()
child.delegate = self
```

Par défaut, un protocol Swift ne permet pas l'implémentation d'une fonction optionnelle. Celles-ci ne peuvent être spécifiées que si votre protocole est marqué avec l'attribut @objc et le modificateur optional .

Par exemple, UITableView implémente le comportement générique d'une vue de table dans iOS, mais l'utilisateur doit implémenter deux classes de délégué appelées UITableViewDelegate et UITableViewDataSource qui implémentent l'apparence et le comportement des cellules spécifiques.

```
@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)
        ...
}
```

Vous pouvez implémenter ce protocole en modifiant votre définition de classe, par exemple:

```
class MyViewController : UIViewController, UITableViewDelegate
```

Toutes les méthodes non marquées comme étant optional dans la définition du protocole (UITableViewDelegate dans ce cas) doivent être implémentées.

Extension de protocole pour une classe conforme spécifique

Vous pouvez écrire l' **implémentation de protocole par défaut** pour une classe spécifique.

```
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"
```

Utilisation du protocole 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")
}
```

Cette structure peut être étendue ailleurs pour ajouter des cas

```
extension NotificationName {
    static let documentationLaunched = NotificationNames(rawValue:
"DocumentationLaunchedNotification")
}
```

Et une interface peut concevoir autour de tout type RawRepresentable ou plus précisément de votre enum struct

```
func post(notification notification: NotificationName) -> Void {
    // use notification.rawValue
}
```

Sur le site d'appel, vous pouvez utiliser un raccourci de syntaxe de point pour les typesafe NotificationName

```
post(notification: .dataFinished)
```

Utilisation de la fonction générique 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
}
```

Protocoles de classe uniquement

Un protocole peut spécifier que seule une classe peut l'implémenter en utilisant le mot class clé class dans sa liste d'héritage. Ce mot-clé doit apparaître avant tout autre protocole hérité de cette liste.

```
protocol ClassOnlyProtocol: class, SomeOtherProtocol {
    // Protocol requirements
}
```

Si un type non-classe tente d'implémenter ClassOnlyProtocol , une erreur de compilation sera générée.

```
struct MyStruct: ClassOnlyProtocol {
    // error: Non-class type 'MyStruct' cannot conform to class protocol 'ClassOnlyProtocol'
}
```

D'autres protocoles peuvent hériter du ClassOnlyProtocol , mais ils auront la même exigence de classe.

```
protocol MyProtocol: ClassOnlyProtocol {
    // ClassOnlyProtocol Requirements
    // MyProtocol Requirements
}

class MySecondClass: MyProtocol {
    // ClassOnlyProtocol Requirements
    // MyProtocol Requirements
}
```

Sémantique de référence des protocoles de classe uniquement

L'utilisation d'un protocole de classe uniquement permet une sémantique de référence lorsque le type conforme est inconnu.

```
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"
}
```

Dans cet exemple, Foo étant un protocole de classe uniquement, l'affectation à la bar est valide car le compilateur sait que foo est un type de classe et qu'il a donc une sémantique de référence.

Si Foo n'était pas un protocole de classe uniquement, une erreur de compilation serait générée - car le type conforme pourrait être un type de valeur , ce qui nécessiterait une annotation var pour pouvoir être muté.

```
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 faibles de type de protocole

Lors de l'application du modificateur weak à une variable de type protocole, ce type de protocole doit être uniquement de classe, car weak ne peut être appliqué qu'à des types de référence.

```
weak var weakReference : ClassOnlyProtocol?
```

Implémentation du protocole Hashable

Les types utilisés dans Sets et Dictionaries(key) doivent être conformes au protocole Hashable qui hérite du protocole Equatable .

Le type personnalisé conforme au protocole Hashable doit implémenter

- Une propriété calculée hashValue
- Définissez l'un des opérateurs d'égalité, à savoir == ou != .

L'exemple suivant implémente le protocole Hashable pour une struct personnalisée:

```
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))
```

Remarque : il n'est pas nécessaire que différentes valeurs du type personnalisé aient des valeurs de hachage différentes, les collisions sont acceptables. Si les valeurs de hachage sont égales, l'opérateur d'égalité sera utilisé pour déterminer la véritable égalité.

Lire Protocoles en ligne: https://riptutorial.com/fr/swift/topic/241/protocoles

Chapitre 52: Réflexion

Syntaxe

- Miroir (reflétant: instance) // Initialise un miroir avec l'objet à réfléchir
- mirror.displayStyle // Style d'affichage utilisé pour les terrains de jeux Xcode
- mirror.description // Représentation textuelle de cette instance, voir CustomStringConvertible
- mirror.subjectType // Retourne le type du sujet réfléchi
- mirror.superclassMirror // Retourne le miroir de la super-classe du sujet réfléchi

Remarques

1. ***Remarques générales:***

Un Mirror est une struct utilisée dans l'introspection d'un objet dans Swift. Sa propriété la plus importante est le tableau des enfants. Un cas d'utilisation possible est de sérialiser une structure pour les Core Data . Cela se fait en convertissant une struct en un NSManagedObject .

2. ***Utilisation de base pour Mirror Remarques:***

La propriété children d'un Mirror est un tableau d'objets enfants provenant de l'objet que l'instance miroir reflète. Un objet child a deux propriétés label et value . Par exemple, un enfant peut être une propriété avec le title et la valeur de Game of Thrones: A Song of Ice and Fire .

Examples

Utilisation de base pour miroir

Créer la classe pour être le sujet du miroir

```
class Project {
    var title: String = ""
    var id: Int = 0
    var platform: String = ""
    var version: Int = 0
    var info: String?
}
```

Créer une instance qui sera effectivement le sujet du miroir. Ici aussi, vous pouvez ajouter des valeurs aux propriétés de la classe Project.

```
let sampleProject = Project()
sampleProject.title = "MirrorMirror"
sampleProject.id = 199
sampleProject.platform = "iOS"
sampleProject.version = 2
sampleProject.info = "test app for Reflection"
```

Le code ci-dessous montre la création de l'instance Mirror. La propriété children du miroir est AnyForwardCollection<Child> où Child est typealias tuple pour la propriété et la valeur de l'objet. Child avait une label: String et 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)")
}
```

Sortie dans Playground ou Console dans Xcode pour la boucle for ci-dessus.

```
title:MirrorMirror
id:199
platform:iOS
version:2
info:Optional("test app for Reflection")
```

Testé sur Playground sur Xcode 8 beta 2

Obtenir le type et les noms des propriétés d'une classe sans avoir à l'instancier

L'utilisation de la classe Swift Mirror fonctionne si vous souhaitez extraire le *nom* , la *valeur* et le *type* (Swift 3: type(of: value) , Swift 2: value.dynamicType) des propriétés d'une **instance** d'une certaine classe.

Si votre classe hérite de NSObject , vous pouvez utiliser la méthode class_copyPropertyList avec property_getAttributes pour rechercher le *nom* et les *types* de propriétés d'une classe, **sans en avoir une instance** . J'ai créé un projet sur Github pour cela, mais voici le code:

```
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
    }
```

Où primitiveDataTypes est un dictionnaire mappant une lettre dans la chaîne d'attribut à un type de valeur:

```
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
    }
```

Il peut extraire le NSObject.Type de toutes les propriétés dont le type de classe hérite de NSObject telles que NSDate (Swift3: Date), NSString (Swift3: String ?) Et NSNumber , mais il est NSNumber dans le type Any (comme vous pouvez le voir type de la valeur du dictionnaire renvoyé par la méthode). Cela est dû aux limitations des value types de value types tels que Int, Int32, Bool. Comme ces types n'héritent pas de NSObject, appeler .self , par exemple, un objet Int- Int.self ne retourne pas NSObject.Type, mais plutôt le type Any . Ainsi, la méthode renvoie Dictionary<String, Any>? et pas Dictionary<String, NSObject.Type>? .

Vous pouvez utiliser cette méthode comme ceci:

```
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'
```

Vous pouvez également essayer de convertir le Any en NSObject.Type , qui réussira pour toutes

les propriétés héritant de NSObject , vous pourrez alors vérifier le type en utilisant l'opérateur standard == :

```
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 vous déclarez cet opérateur personnalisé == :

```
func ==(rhs: Any, lhs: Any) -> Bool {
    let rhsType: String = "\(rhs)"
    let lhsType: String = "\(lhs)"
    let same = rhsType == lhsType
    return same
}
```

Vous pouvez même vérifier le type de type de value types comme ceci:

```
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'")
        }
    }
}
```

LIMITES Cette solution ne fonctionne pas lorsque les value types sont optionnels. Si vous avez déclaré une propriété dans votre sous-classe NSObject comme ceci: var myOptionalInt: Int? , le code ci-dessus ne trouvera pas cette propriété car la méthode class_copyPropertyList ne contient pas de types de valeur facultatifs.

Lire Réflexion en ligne: https://riptutorial.com/fr/swift/topic/1201/reflexion

Chapitre 53: RxSwift

Examples

Bases RxSwift

La FRP, ou programmation réactive fonctionnelle, comporte certains termes de base que vous devez connaître.

Chaque donnée peut être représentée comme Observable , qui est un flux de données asynchrone. La puissance de FRP est en représentation des événements synchrones et asynchrones en tant que flux, Observable et fournissant la même interface pour travailler avec lui.

Habituellement, Observable détient plusieurs (ou aucun) événements qui contiennent les événements de date - .Next , puis il peut être terminé avec succès (.Success) ou avec une erreur (.Error).

Jetons un coup d'oeil au diagramme de marbre suivant:

```
--(1)--(2)--(3)|-->
```

Dans cet exemple, il y a un flux de valeurs Int . Au fil du temps, trois événements .Next se sont produits, puis le flux s'est terminé avec succès.

```
--X->
```

Le diagramme ci-dessus montre un cas où aucune donnée n'a été émise et l'événement .Error termine l' Observable .

Avant de continuer, il existe des ressources utiles:

1. RxSwift . Regardez des exemples, lisez des documents et commencez.
2. RxSwift Slack room dispose de quelques canaux pour résoudre les problèmes d'éducation.
3. Jouez avec RxMarbles pour savoir ce que fait l'opérateur et lequel est le plus utile dans votre cas.
4. Regardez cet exemple , explorez le code par vous-même.

Créer des observables

RxSwift propose de nombreuses façons de créer une Observable , jetons un coup d'œil:

```
import RxSwift

let intObservale = Observable.just(123) // Observable<Int>
let stringObservale = Observable.just("RxSwift") // Observable<String>
let doubleObservale = Observable.just(3.14) // Observable<Double>
```

Ainsi, les observables sont créés. Ils ne contiennent qu'une valeur et se terminent ensuite avec succès. Néanmoins, rien ne se passe après sa création. Pourquoi?

Travailler avec Observable s en deux étapes: vous **observez** quelque chose pour *créer* un flux, puis vous vous **abonnez** au flux ou le **liez** à quelque chose pour *interagir* avec lui.

```
Observable.just(12).subscribe {
    print($0)
}
```

La console imprimera:

```
.Next(12)
.Completed()
```

Et si je ne m'intéressais qu'au travail avec les données, qui ont lieu dans les événements .Next , j'utiliserais l'opérateur subscribeNext :

```
Observable.just(12).subscribeNext {
    print($0) // prints "12" now
}
```

Si je veux créer une observable de plusieurs valeurs, j'utilise différents opérateurs:

```
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.
```

Et enfin, peut-être que je veux une Observable qui fait du travail. Par exemple, il est pratique d'emballer une opération réseau dans Observable<SomeResultType> . Jetons un coup d'œil à ce que l'on peut réaliser:

```
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.
}
```

Jeter

Une fois l'abonnement créé, il est important de gérer sa désallocation correcte.

Les docs nous ont dit que

> Si une séquence se termine dans un temps fini, ne pas appeler disposer ou ne pas utiliser addDisposableTo (disposeBag) ne provoquera aucune fuite de ressource permanente. Cependant, ces ressources seront utilisées jusqu'à la fin de la séquence, soit en finissant la production d'éléments, soit en renvoyant une erreur.

Il existe deux manières de désallouer les ressources.

1. Utilisation de l' disposeBag s et addDisposableTo .
2. Utilisation de l'opérateur takeUntil .

Dans le premier cas, vous passez manuellement l'abonnement à l'objet DisposeBag , qui efface correctement toute la mémoire prise.

```
let bag = DisposeBag()
Observable.just(1).subscribeNext {
    print($0)
}.addDisposableTo(bag)
```

Vous n'avez pas vraiment besoin de créer des DisposeBag dans chaque classe que vous créez, jetez un coup d'œil au *projet de la communauté RxSwift* nommé NSObject + Rx . En utilisant le framework, le code ci-dessus peut être réécrit comme suit:

```
Observable.just(1).subscribeNext {
    print($0)
}.addDisposableTo(rx_disposeBag)
```

Dans le second cas, si le temps d'abonnement coïncide avec la durée de vie de l'objet self , il est possible d'implémenter l'élimination en utilisant takeUntil(rx_deallocated) :

```
let _ = sequence
    .takeUntil(rx_deallocated)
    .subscribe {
        print($0)
    }
```

Fixations

```
Observable.combineLatest(firstName.rx_text, lastName.rx_text) { $0 + " " + $1 }
.map { "Greetings, \($0)" }
.bindTo(greetingLabel.rx_text)
```

En utilisant l'opérateur combineLatest chaque fois qu'un élément est émis par l'une des deux Observables , combinez le dernier élément émis par chaque Observable . Ainsi, nous combinons le résultat des deux nouveaux messages de création de UITextField avec le texte "Greetings, \($0)" utilisant une interpolation de chaîne pour se lier plus tard au texte d'un UILabel .

Nous pouvons lier des données à n'importe quel UITableView et UICollectionView de manière très simple:

```
viewModel
.rows
.bindTo(resultsTableView.rx_itemsWithCellIdentifier("WikipediaSearchCell", cellType:
WikipediaSearchCell.self)) { (_, viewModel, cell) in
    cell.title = viewModel.title
    cell.url = viewModel.url
}
.addDisposableTo(disposeBag)
```

C'est un wrapper Rx autour de la méthode de source de données cellForRowAtIndexPath . Et Rx prend également en charge l'implémentation de numberOfRowsAtIndexPath , qui est une méthode requise dans un sens traditionnel, mais vous n'avez pas à l'implémenter ici, c'est fait.

RxCocoa et ControlEvents

RxSwift fournit non seulement les moyens de contrôler vos données, mais également de représenter les actions de l'utilisateur de manière réactive.

RxCocoa contient tout ce dont vous avez besoin. Il encapsule la plupart des propriétés des composants de l'interface utilisateur dans Observable , mais pas vraiment. Il existe des Observable mis à niveau, appelés ControlEvent s (qui représentent des événements) et ControlProperties (qui représentent des propriétés, des surprises!). Ces objets contiennent des flux Observable sous le capot, mais ont également des nuances:

- Il n'échoue jamais, donc pas d'erreurs.
- Il Complete séquence sur le contrôle en cours de désallocation.
- Il délivre des événements sur le thread principal (MainScheduler.instance).

Fondamentalement, vous pouvez travailler avec eux comme d'habitude:

```
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.
```

Ceci est très important à utiliser: tant que vous utilisez Rx, oubliez les éléments de @IBAction , tout ce dont vous avez besoin pour vous @IBAction et configurer en même temps. Par exemple, la méthode viewDidLoad de votre contrôleur de vue est un bon candidat pour décrire le fonctionnement des composants de l'interface utilisateur.

Ok, un autre exemple: supposons que nous ayons un champ de texte, un bouton et une étiquette. Nous voulons **valider le texte** dans le champ de **texte** lorsque nous tapons **sur** le bouton et **afficher** les résultats dans l'étiquette. Ouais, semble être une autre tâche de validation de courrier électronique, hein?

Tout d'abord, nous récupérons le bouton button.rx_tap ControlEvent:

```
----()-----------------------()----->
```

Ici, les parenthèses vides indiquent les touches utilisateur. Ensuite, nous prenons ce qui est écrit dans textField avec l'opérateur withLatestFrom (regardez-le ici , imaginez que le flux supérieur représente les taps des utilisateurs, le bas représente le texte dans le champ de texte).

```
button.rx_tap.withLatestFrom(textField.rx_text)

----("")--------------------("123")--->
//  ^ tap   ^ i wrote 123    ^ tap
```

Bien, nous avons un flux de chaînes à valider, émis uniquement lorsque nous devons valider.

Tout Observable a des opérateurs aussi familiers que map ou filter , nous prendrons la map pour valider le texte. Créez vous-même la fonction validateEmail , utilisez toutes les expressions que vous souhaitez.

```
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)
```

Terminé! Si vous avez besoin de plus de logique personnalisée (comme afficher des vues d'erreur en cas d'erreur, effectuer une transition vers un autre écran en cas de succès ...), abonnez-vous simplement au flux Bool final et écrivez-le.

Lire RxSwift en ligne: https://riptutorial.com/fr/swift/topic/4890/rxswift

Chapitre 54: Structs

Examples

Bases de Structs

```
struct Repository {
    let identifier: Int
    let name: String
    var description: String?
}
```

Cela définit une structure de Repository avec trois propriétés stockées, un identifier entier, un name chaîne et une description chaîne facultative. L' identifier et le name sont des constantes, car ils ont été déclarés en utilisant le mot let clé let . Une fois définies lors de l'initialisation, elles ne peuvent plus être modifiées. La description est une variable. La modifier met à jour la valeur de la structure.

Les types de structure reçoivent automatiquement un initialiseur membre s'ils ne définissent aucun de leurs initialiseurs personnalisés. La structure reçoit un initialiseur membre, même si des propriétés stockées ne sont pas définies par défaut.

Repository contient trois propriétés stockées dont seule la description a une valeur par défaut (nil). De plus, il ne définit aucun initialiseur, il reçoit donc un initialiseur membre par code:

```
let newRepository = Repository(identifier: 0, name: "New Repository", description: "Brand New Repository")
```

Les structures sont des types de valeur

Contrairement aux classes, qui sont transmises par référence, les structures sont transmises via la copie:

```
first = "Hello"
second = first
first += " World!"
// first == "Hello World!"
// second == "Hello"
```

String est une structure, elle est donc copiée lors de l'affectation.

Les structures ne peuvent pas non plus être comparées à l'aide de l'opérateur d'identité:

```
window0 === window1 // works because a window is a class instance
"hello" === "hello" // error: binary operator '===' cannot be applied to two 'String' operands
```

Toutes les deux instances de structure sont réputées identiques si elles sont égales.

Collectivement, ces traits qui différencient les structures des classes sont ce qui fait que les types de valeur des structures.

Mutant d'une structure

Une méthode d'une structure qui modifie la valeur de la structure doit être préfixée avec le mot-clé en mutating

```
struct Counter {
    private var value = 0

    mutating func next() {
        value += 1
    }
}
```

Lorsque vous pouvez utiliser des méthodes de mutation

Les méthodes de mutating ne sont disponibles que sur les valeurs struct à l'intérieur des variables.

```
var counter = Counter()
counter.next()
```

Lorsque vous ne pouvez PAS utiliser de méthodes de mutation

D'un autre côté, les méthodes de mutating NE sont PAS disponibles sur les valeurs de structure à l'intérieur des constantes.

```
let counter = Counter()
counter.next()
//  error: cannot use mutating member on immutable value: 'counter' is a 'let' constant
```

Les structures ne peuvent pas hériter

Contrairement aux classes, les structures ne peuvent pas hériter:

```
class MyView: NSView { }  // works

struct MyInt: Int { } // error: inheritance from non-protocol type 'Int'
```

Les structures peuvent toutefois adopter des protocoles:

```
struct Vector: Hashable { ... }  // works
```

Accéder aux membres de struct

Dans Swift, les structures utilisent une simple «syntaxe à points» pour accéder à leurs membres.

Par exemple:

```
struct DeliveryRange {
  var range: Double
  let center: Location
}
let storeLocation = Location(latitude: 44.9871,
                             longitude: -93.2758)
var pizzaRange = DeliveryRange(range: 200,
                               center: storeLocation)
```

Vous pouvez accéder à (imprimer) la plage comme ceci:

```
print(pizzaRange.range) // 200
```

Vous pouvez même accéder aux membres de membres en utilisant la syntaxe à point:

```
print(pizzaRange.center.latitude) // 44.9871
```

De la même manière que vous pouvez lire des valeurs avec une syntaxe à point, vous pouvez également les affecter.

```
pizzaRange.range = 250
```

Lire Structs en ligne: https://riptutorial.com/fr/swift/topic/255/structs

Chapitre 55: Swift HTTP server par Kitura

Introduction

Serveur Swift avec Kitura

Kitura est un framework web écrit en swift qui est créé pour les services Web. Il est très facile à configurer pour les requêtes HTTP. Pour l'environnement, il faut soit OS X avec XCode installé, soit Linux avec swift 3.0.

Examples

Bonjour application mondiale

Configuration

Tout d'abord, créez un fichier appelé Package.swift. C'est le fichier qui indique au compilateur rapide où se trouvent les bibliothèques. Dans cet exemple mondial, nous utilisons les repos GitHub. Nous avons besoin de Kitura et HeliumLogger . Placez le code suivant dans Package.swift. Il spécifiait le nom du projet comme *kitura-helloworld* et également les URL de dépendance.

```
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) ] )
```

Ensuite, créez un dossier appelé Sources. À l'intérieur, créez un fichier appelé main.swift. C'est le fichier que nous implémentons toute la logique pour cette application. Entrez le code suivant dans ce fichier principal.

Importer des bibliothèques et activer la journalisation

```
import Kitura
import Foundation
import HeliumLogger

HeliumLogger.use()
```

Ajout d'un routeur Le routeur spécifie un chemin, un type, etc. de la requête HTTP. Nous ajoutons ici un gestionnaire de requêtes GET qui imprime *Hello world* , puis une post-requête qui lit le texte brut de la requête et le renvoie ensuite.

```
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()
}
```

Spécifiez un port pour exécuter le service

```
let port = 8080
```

Liez le routeur et le port ensemble et ajoutez-les en tant que service HTTP

```
Kitura.addHTTPServer(onPort: port, with: router)
Kitura.run()
```

Exécuter

Accédez au dossier racine avec le fichier Package.swift et le dossier Resources. Exécutez la commande suivante. Le compilateur Swift télécharge automatiquement les ressources mentionnées dans Package.swift dans le dossier Packages, puis compile ces ressources avec main.swift.

```
swift build
```

Lorsque la construction est terminée, l'exécutable sera placé à cet endroit. Double-cliquez sur cet exécutable pour démarrer le serveur.

```
.build/debug/kitura-helloworld
```

Valider

Ouvrez un navigateur, tapez localhost:8080/get as url et appuyez sur Entrée. La page de salut de monde devrait sortir.

Ouvrez une application de requête HTTP, publiez du texte brut sur localhost:8080/post . La chaîne de réponse affichera le texte entré correctement.

localhost:8080/post

POST localhost:8080/post

Authorization Headers (1) Body Pre-request Scr

form-data x-www-form-urlencoded raw bin

```
1 Some text
```

Body Cookies Headers (4) Tests

```
1 Value Some text received.
```

Lire Swift HTTP server par Kitura en ligne: https://riptutorial.com/fr/swift/topic/10690/swift-http-server-par-kitura

Chapitre 56: Tableaux

Introduction

Le tableau est un type de collection à accès aléatoire ordonné. Les tableaux constituent l'un des types de données les plus couramment utilisés dans une application. Nous utilisons le type Array pour contenir des éléments d'un seul type, le type Element du tableau. Un tableau peut stocker n'importe quel type d'éléments - des entiers aux chaînes de caractères.

Syntaxe

- Array <Element> // Le type d'un tableau avec des éléments de type Element
- [Element] // Sucre syntaxique pour le type d'un tableau avec des éléments de type Element
- [element0, element1, element2, ... elementN] // Un littéral tableau
- [Element] () // Crée un nouveau tableau vide de type [Element]
- Array (count: repeatValue :) // Crée un tableau d'éléments count , chacun initialisé à repeatedValue
- Array (_ :) // Crée un tableau à partir d'une séquence arbitraire

Remarques

Les tableaux sont une *collection ordonnée* de valeurs. Les valeurs peuvent se répéter mais *doivent* être du même type.

Examples

Sémantique de la valeur

Copier un tableau copiera tous les éléments dans le tableau d'origine.

Changer le nouveau tableau *ne changera pas* le tableau d'origine.

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

Les tableaux copiés partageront le même espace en mémoire que l'original jusqu'à ce qu'ils soient modifiés. Il en résulte un problème de performance lorsque le tableau copié reçoit son propre espace en mémoire car il est modifié pour la première fois.

Bases de tableaux

Array est un type de collection ordonnée dans la bibliothèque standard Swift. Il fournit un accès aléatoire O (1) et une réallocation dynamique. Array est un type générique , de sorte que le type de valeurs qu'il contient est connu à la compilation.

Comme Array est un type de valeur , sa mutabilité est définie par le fait qu'il soit annoté en tant que var (mutable) ou let (immutable).

Le type [Int] (signification: un tableau contenant Int s) est du sucre syntaxique pour Array<T> .

En savoir plus sur les tableaux dans le langage de programmation Swift .

Tableaux vides

Les trois déclarations suivantes sont équivalentes:

```
// 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
```

Littéraux de tableau

Un littéral de tableau est écrit avec des crochets entourant les éléments séparés par des virgules:

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

Le compilateur peut généralement déduire le type d'un tableau basé sur les éléments du littéral, mais les **annotations de type** explicite peuvent remplacer le paramètre par défaut:

```
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
```

Tableaux avec valeurs répétées

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

Création de tableaux à partir d'autres séquences

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

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

Tableaux multidimensionnels

Dans Swift, un tableau multidimensionnel est créé en imbriquant des tableaux: un tableau à deux dimensions de Int est [[Int]] (ou Array<Array<Int>>).

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

Pour créer un tableau multidimensionnel de valeurs répétées, utilisez les appels imbriqués de l'initialiseur de tableau:

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

Accéder aux valeurs du tableau

Les exemples suivants utiliseront ce tableau pour démontrer l'accès aux valeurs

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

Pour accéder à une valeur à un index connu, utilisez la syntaxe suivante:

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

Remarque: La valeur de l' *index deux est la troisième valeur* du Array . Array utilisent un *index basé sur zéro,* ce qui signifie que le premier élément du Array est à l'index 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
```

Accédez à un sous-ensemble d'un Array aide du filtre:

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

Les filtres peuvent avoir des conditions complexes comme le filtrage des nombres pairs uniquement:

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

Il est également possible de retourner l'index d'une valeur donnée, en retournant nil si la valeur n'a pas été trouvée.

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

Il existe des méthodes pour la valeur première, dernière, maximale ou minimale dans un Array . Ces méthodes renvoient nil si le Array est vide.

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

Méthodes utiles

Détermine si un tableau est vide ou renvoie sa taille

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

Inverser un tableau **Remarque: Le résultat n'est pas appliqué au tableau sur lequel la méthode est appelée et doit être placé dans sa propre variable.**

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

Modification de valeurs dans un tableau

Il y a plusieurs façons d'ajouter des valeurs à un tableau

```
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]
```

et supprimer des valeurs d'un tableau

```
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]
```

Tri d'un tableau

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

Créer un nouveau tableau trié

Comme Array est conforme à SequenceType , nous pouvons générer un nouveau tableau des éléments triés en utilisant une méthode de tri intégrée.

2.1 2.2

Dans Swift 2, cela se fait avec la méthode sort() .

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

3.0

A partir de Swift 3, il a été renommé en sorted() .

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

Trier un tableau existant en place

Comme Array est conforme à MutableCollectionType , nous pouvons trier ses éléments en place.

2.1 2.2

Dans Swift 2, cela se fait en utilisant la méthode sortInPlace() .

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

3.0

Depuis Swift 3, il a été renommé pour sort() .

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

> Remarque: Pour utiliser les méthodes ci-dessus, les éléments doivent être conformes au protocole Comparable .

Trier un tableau avec un ordre personnalisé

Vous pouvez également trier un tableau en utilisant une fermeture pour définir si un élément doit être ordonné avant un autre, ce qui ne se limite pas aux tableaux où les éléments doivent être Comparable . Par exemple, il n'est pas logique qu'un Landmark de Landmark soit Comparable , mais vous pouvez toujours trier un tableau de points de repère par hauteur ou par nom.

```
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}
```

> Remarque: La comparaison de chaînes peut donner des résultats inattendus si les chaînes sont incohérentes. Voir Tri d'un tableau de chaînes .

Transformer les éléments d'un tableau avec map (_ :)

Comme Array est conforme à SequenceType , nous pouvons utiliser map(_:) pour transformer un tableau de A en un tableau de B utilisant une fermeture de type (A) throws -> B

Par exemple, nous pourrions l'utiliser pour transformer un tableau de Int s en un tableau de String comme suit:

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

map(_:) va parcourir le tableau, en appliquant la fermeture donnée à chaque élément. Le résultat de cette fermeture sera utilisé pour remplir un nouveau tableau avec les éléments transformés.

Puisque String a un initialiseur qui reçoit un Int nous pouvons également utiliser cette syntaxe plus claire:

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

Une map(_:) transform n'a pas besoin de changer le type du tableau - par exemple, elle pourrait aussi être utilisée pour multiplier un tableau de Int s par deux:

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

Extraire des valeurs d'un type donné à partir d'un tableau avec flatMap (_ :)

Les things Array contiennent des valeurs de type Any .

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

Nous pouvons extraire les valeurs d'un type donné et créer un nouveau tableau de ce type spécifique. Disons que nous voulons extraire tous les Int(s) et les placer dans un tableau Int Array de manière sûre.

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

Maintenant, les numbers sont définis comme [Int] . La fonction flatMap tous les éléments nil et le résultat ne contient donc que les valeurs suivantes:

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

Filtrage d'un tableau

Vous pouvez utiliser la méthode filter(_:) sur SequenceType pour créer un nouveau tableau contenant les éléments de la séquence qui satisfont un prédicat donné, qui peut être fourni en tant que fermeture .

Par exemple, filtrer les nombres pairs d'un [Int] :

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

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

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

Filtrer un [Person] , où son âge est inférieur à 30:

```
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)]
```

Filtrage de la transformation Nil d'une matrice avec flatMap (_ :)

Vous pouvez utiliser flatMap(_:) de manière similaire à map(_:) afin de créer un tableau en appliquant une transformation aux éléments d'une séquence.

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

La différence avec cette version de flatMap(_:) est qu'elle attend que la fermeture de la transformation renvoie une valeur optionnelle T? pour chacun des éléments. Il va ensuite déballer en toute sécurité chacune de ces valeurs optionnelles, en filtrant la valeur nil - ce qui donnera un tableau de [T] .

Par exemple, vous pouvez le faire afin de transformer un [String] en un [Int] utilisant l' initialiseur de String disponible d' Int , en filtrant tous les éléments qui ne peuvent pas être convertis:

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

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

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

Vous pouvez également utiliser la flatMap(_:) à filtrer le nil afin de convertir simplement un tableau de flatMap(_:) optionnels en un tableau de données non optionnelles:

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

let numbers = optionalNumbers.flatMap { $0 }

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

Inscrire un tableau avec une plage

On peut extraire une série d'éléments consécutifs d'un tableau en utilisant une plage.

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

L'indexation d'un tableau avec une plage renvoie un ArraySlice . C'est une sous-séquence du tableau.

Dans notre exemple, nous avons un tableau de chaînes, nous ArraySlice<String> donc ArraySlice<String> .

Bien qu'un ArraySlice soit conforme à CollectionType et puisse être utilisé avec un sort , un filter , etc., il ne sert pas au stockage à long terme mais à des calculs transitoires: il doit être reconverti en un tableau dès que vous avez fini de l'utiliser.

Pour cela, utilisez l'initialiseur Array() :

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

Pour résumer dans un exemple simple sans étapes intermédiaires:

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

Regroupement des valeurs de tableau

Si nous avons une structure comme celle-ci

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

et un tableau 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)
]
```

nous pouvons regrouper les boîtes par la propriété thingsInside pour obtenir un Dictionary où la key est le nombre de choses et la valeur un tableau de boîtes.

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

Maintenant regroupé est un [Int:[Box]] et a le contenu suivant

```
[
    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)]
]
```

Aplatir le résultat d'une transformation Array avec flatMap (_ :)

En plus d'être en mesure de créer un tableau de filtrage de nil à partir des éléments transformées d'une séquence, il existe également une version de flatMap(_:) qui prévoit la

transformation de fermeture pour retourner une séquence S .

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

Chaque séquence de la transformation sera concaténée, résultant en un tableau contenant les éléments combinés de chaque séquence - [S.Generator.Element] .

Combiner les caractères dans un tableau de chaînes

Par exemple, nous pouvons l'utiliser pour prendre un tableau de chaînes principales et combiner leurs caractères en un seul tableau:

```
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"]"
```

Briser l'exemple ci-dessus:

1. primes est un [String] (comme un tableau est une séquence, nous pouvons appeler flatMap(_:)).
2. La fermeture de transformation prend l'un des éléments de primes , une String (Array<String>.Generator.Element).
3. La fermeture retourne alors une séquence de type String.CharacterView .
4. Le résultat est alors un tableau contenant les éléments combinés de toutes les séquences de chacun des appels de clôture de transformation - [String.CharacterView.Generator.Element] .

Aplatir un tableau multidimensionnel

Comme flatMap(_:) concaténera les séquences renvoyées par les appels de clôture de transformation, il peut être utilisé pour aplatir un tableau multidimensionnel, tel qu'un tableau 2D en un tableau 1D, un tableau 3D en un tableau 2D, etc.

Cela peut se faire simplement en retournant l'élément donné $0 (un tableau imbriqué) dans la fermeture:

```
// 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]
```

Trier un tableau de chaînes

3.0

Le moyen le plus simple est d'utiliser sorted() :

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

ou sort()

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

Vous pouvez passer une fermeture en argument pour le tri:

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

Syntaxe alternative avec une clôture finale:

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

Mais il y aura des résultats inattendus si les éléments du tableau ne sont pas cohérents:

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

Pour résoudre ce problème, triez sur une version minuscule des éléments:

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

Ou import Foundation et utilisez les méthodes de comparaison de NSString telles que caseInsensitiveCompare :

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

Vous pouvez également utiliser localizedCaseInsensitiveCompare , qui peut gérer les signes diacritiques.

Pour trier correctement Strings par la valeur *numérique* qu'ils contiennent, utilisez compare avec l'option .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"]
```

Aplatir paresseusement un tableau multidimensionnel avec flatten ()

Nous pouvons utiliser flatten() pour réduire paresseusement l'imbrication d'une séquence multidimensionnelle.

Par exemple, la mise à plat paresseuse d'un tableau 2D en un tableau 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
```

Dans l'exemple ci-dessus, flatten() retournera un FlattenBidirectionalCollection , qui appliquera paresseusement l'aplatissement du tableau. La fonction contains(_:) ne nécessitera donc que l' array2D des deux premiers tableaux imbriqués de array2D , car il court-circuitera lors de la recherche de l'élément souhaité.

Combiner les éléments d'un tableau avec réduire (_: combiner :)

reduce(_:combine:) peut être utilisé pour combiner les éléments d'une séquence en une seule valeur. Il faut une valeur initiale pour le résultat, ainsi qu'une fermeture à appliquer à chaque élément - qui renverra la nouvelle valeur accumulée.

Par exemple, nous pouvons l'utiliser pour additionner un tableau de nombres:

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

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

print(sum) // 36
```

Nous passons 0 dans la valeur initiale, car c'est la valeur initiale logique pour une sommation. Si nous passons dans une valeur de N , la sum résultante serait N + 36 . La fermeture passée à reduce a deux arguments. accumulator est la valeur accumulée actuelle, à laquelle est affectée la valeur renvoyée par la fermeture à chaque itération. element est l'élément actuel de l'itération.

Comme dans cet exemple, nous passons une fermeture (Int, Int) -> Int à reduce , qui produit simplement l'ajout des deux entrées - nous pouvons en fait passer directement l'opérateur + , car les opérateurs sont des fonctions dans Swift:

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

Suppression d'un élément d'un tableau sans connaître son index

Généralement, si nous voulons supprimer un élément d'un tableau, nous devons connaître son index afin de pouvoir le supprimer facilement en utilisant la fonction remove(at:) .

Mais que faire si on ne connait pas l'index mais on connait la valeur de l'élément à supprimer!

Voici donc la simple extension à un tableau qui nous permettra de supprimer facilement un élément du tableau sans connaître son index:

Swift3

```
extension Array where Element: Equatable {

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

par exemple

```
    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!
```

Aussi , si, par erreur, nous avons fait quelque chose comme ceci: array.remove(25) par exemple , nous avons fourni la valeur avec différents types de données, compilateur génère une erreur mentioning-
cannot convert value to expected argument type

Trouver l'élément minimum ou maximum d'un tableau

2.1 2.2

Vous pouvez utiliser les minElement() et maxElement() pour rechercher l'élément minimum ou maximum dans une séquence donnée. Par exemple, avec un tableau de nombres:

```
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, les méthodes ont été renommées respectivement min() et max() :

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

Les valeurs renvoyées par ces méthodes sont facultatives pour refléter le fait que le tableau pourrait être vide. Si tel est le cas, nil ne sera renvoyée.

> Remarque: Les méthodes ci-dessus nécessitent que les éléments soient conformes au protocole Comparable .

Trouver l'élément minimum ou maximum avec un ordre personnalisé

Vous pouvez également utiliser les méthodes ci-dessus avec une fermeture personnalisée, en définissant si un élément doit être commandé avant un autre, ce qui vous permet de trouver l'élément minimum ou maximum dans un tableau où les éléments ne sont pas nécessairement Comparable .

Par exemple, avec un tableau de vecteurs:

```
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 }
```

Accéder aux indices en toute sécurité

En ajoutant l'extension suivante aux tableaux, vous pouvez accéder aux indices sans savoir si l'index est compris dans les limites.

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

Exemple:

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

Comparaison de 2 tableaux avec zip

La fonction zip accepte 2 paramètres de type SequenceType et retourne une Zip2Sequence où chaque élément contient une valeur de la première séquence et une de la seconde.

Exemple

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

nomsAndAnimals contient maintenant les valeurs suivantes

séquence1	séquence1
1	"Dog"
2	"Cat"
3	"Tiger"

Ceci est utile lorsque vous souhaitez effectuer une sorte de comparaison entre le n-ième élément de chaque tableau.

Exemple

Étant donné 2 tableaux de Int(s)

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

vous voulez vérifier si chaque valeur dans list1 est le double de la valeur associée dans list0
.

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

Lire Tableaux en ligne: https://riptutorial.com/fr/swift/topic/284/tableaux

Chapitre 57: Travailler avec C et Objective-C

Remarques

Pour plus d'informations, consultez la documentation d'Apple sur l' utilisation de Swift avec Cocoa et Objective-C .

Examples

Utiliser les classes Swift du code Objective-C

Dans le même module

Dans un module nommé " **MyModule** ", Xcode génère un en-tête nommé **MyModule-Swift.h** qui expose les classes Swift publiques à Objective-C. Importez cet en-tête pour utiliser les classes 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 {
    // ...
}
```

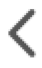
```
// MyViewController.m in MyApp

#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
```

Paramètres de construction pertinents:

- **Nom de l'en-tête de l'interface générée par Objective-C** : contrôle le nom de l'en-tête Obj-C généré.
- **Installez Objective-C Compatibility Header** : si l'en-tête -Swift.h doit être un en-tête public (pour les cibles du framework).



Dans un autre module

Utiliser **@import MyFramework;** importe l'ensemble du module, y compris les interfaces Obj-C, aux classes Swift (si le paramètre de construction susmentionné est activé).

Utilisation de classes Objective-C à partir du code Swift

Si MyFramework contient des classes Objective-C dans ses en-têtes publiques (et dans l'en-tête parapluie), il **import MyFramework** pour les utiliser depuis Swift.

Passerelles

Un en- **tête de pontage** rend les déclarations Objective-C et C supplémentaires visibles pour le code Swift. Lors de l'ajout de fichiers de projet, Xcode peut proposer de créer automatiquement un en-tête de pontage:

Would you like to configure an Objectiv

Adding this file to MyApp will create a mixed Sw
you like Xcode to automatically configure a brid
be accessed by both languages?

Cancel   Don't Create

Pour en créer un manuellement, modifiez le paramètre de génération **Objective-C Bridging Header** :

▼ Swift Compiler - Code Generation

Setting

Disable Safety Checks

Install Objective-C Compatibility Header

▶ Objective-C Bridging Header

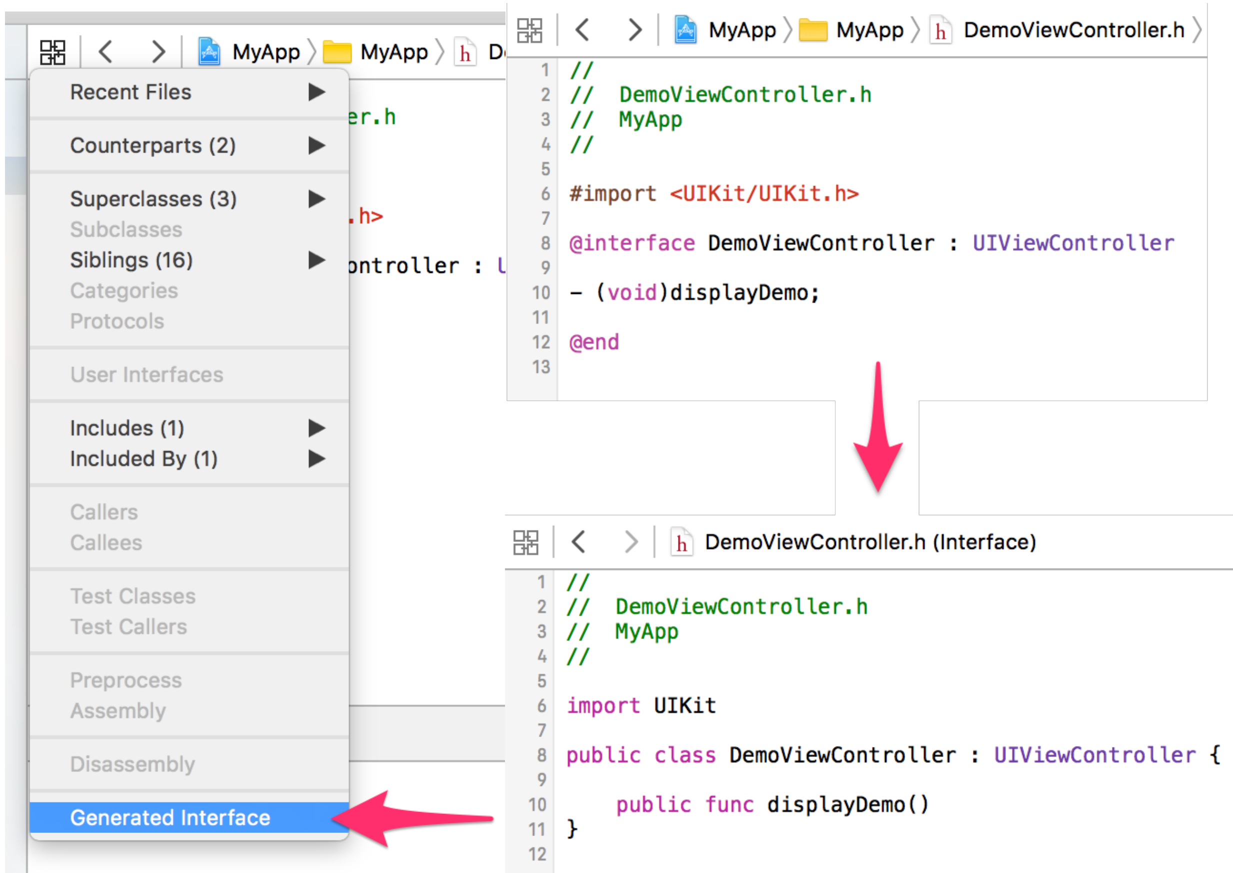
Objective-C Generated Interface Header Name

A l'intérieur de l'en-tête de pontage, importez les fichiers nécessaires à l'utilisation du code:

```
// MyApp-Bridging-Header.h
#import "MyClass.h"  // allows code in this module to use MyClass
```

Interface générée

Cliquez sur le bouton Éléments associés (ou appuyez sur ^1), puis sélectionnez **Interface générée** pour voir l'interface Swift générée à partir d'un en-tête Objective-C.



Spécifiez un en-tête de pontage pour swiftc

Le -import-objc-header spécifie un en-tête pour swiftc à importer:

```
// defs.h
struct Color {
    int red, green, blue;
};

#define MAX_VALUE 255
```

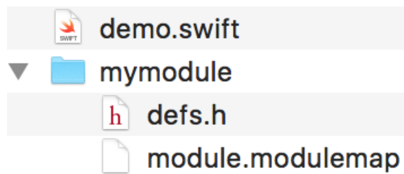
```
// demo.swift
extension Color: CustomStringConvertible {  // extension on a C struct
    public var description: String {
        return "Color(red: \(red), green: \(green), blue: \(blue))"
    }
}
print("MAX_VALUE is: \(MAX_VALUE)")  // C macro becomes a constant
let color = Color(red: 0xCA, green: 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)
```

Utiliser une carte de module pour importer des en-têtes C

Une carte de module peut simplement import mymodule en le configurant pour lire les fichiers d'en-tête C et les faire apparaître comme des fonctions Swift.

Placez un fichier nommé module.modulemap dans un répertoire nommé mymodule :



Dans le fichier de carte du module:

```
// mymodule/module.modulemap
module mymodule {
    header "defs.h"
}
```

Puis import le module:

```
// demo.swift
import mymodule
print("Empty color: \(Color())")
```

Utilisez l'indicateur de -I *directory* pour indiquer à swiftc où trouver le module:

```
swiftc -I . demo.swift   # "-I ." means "search for modules in the current directory"
```

Pour plus d'informations sur la syntaxe de la carte de module, consultez la documentation de Clang sur les cartes de module .

Interaction fine entre Objective-C et Swift

Lorsqu'une API est marquée avec NS_REFINED_FOR_SWIFT , elle sera préfixée par deux NS_REFINED_FOR_SWIFT soulignement (__) lors de son importation dans Swift:

```
@interface MyClass : NSObject
- (NSInteger)indexOfObject:(id)obj NS_REFINED_FOR_SWIFT;
@end
```

L' interface générée ressemble à ceci:

```
public class MyClass : NSObject {
    public func __indexOfObject(obj: AnyObject) -> Int
}
```

Vous pouvez maintenant **remplacer l'API** par une extension plus "Swifty". Dans ce cas, nous pouvons utiliser une valeur de retour facultative , en filtrant 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
}
```

Dans la plupart des cas , vous pouvez limiter ou non un argument à une fonction Objective-C pourrait être nil . Ceci est fait en utilisant le mot clé _Nonnull , qui qualifie tout pointeur ou référence de bloc:

```
void
doStuff(const void *const _Nonnull data, void (^_Nonnull completion)())
{
    // complex asynchronous code
}
```

Avec cela écrit, le compilateur doit émettre une erreur chaque fois que nous essayons de passer nil à cette fonction à partir de notre code Swift:

```
doStuff(
    nil,  // error: nil is not compatible with expected argument type 'UnsafeRawPointer'
    nil)  // error: nil is not compatible with expected argument type '() -> Void'
```

Le contraire de _Nonnull est _Nullable , ce qui signifie qu'il est acceptable de passer nil dans cet argument. _Nullable est aussi la valeur par défaut. Cependant, le spécifier explicitement permet un code plus auto-documenté et à l'épreuve du futur.

Pour aider le compilateur à optimiser votre code, vous pouvez également spécifier si le bloc est échappé:

```
void
callNow(__attribute__((noescape)) void (^_Nonnull f)())
{
    // f is not stored anywhere
}
```

Avec cet attribut, nous promettons de ne pas enregistrer la référence de bloc et de ne pas appeler le bloc une fois l'exécution de la fonction terminée.

Utilisez la bibliothèque standard C

L'interopérabilité C de Swift vous permet d'utiliser des fonctions et des types de la bibliothèque standard C.

Sous Linux, la bibliothèque standard C est exposée via le module Glibc ; sur les plateformes Apple, cela s'appelle 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
```

Lire Travailler avec C et Objective-C en ligne:
https://riptutorial.com/fr/swift/topic/421/travailler-avec-c-et-objective-c

Chapitre 58: Tuples

Introduction

Un type de tuple est une liste de types séparés par des virgules, entre parenthèses.

Cette liste de types peut également avoir le nom des éléments et utiliser ces noms pour faire référence aux valeurs des éléments individuels.

Un nom d'élément consiste en un identifiant suivi immédiatement de deux points (:).

Usage commun -

On peut utiliser un type de tuple comme type de retour d'une fonction pour permettre à la fonction de renvoyer un seul tuple contenant plusieurs valeurs

Remarques

Les tuples sont considérés comme des types de valeur. Vous trouverez plus d'informations sur les tuples dans la documentation:
developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.

Examples

Que sont les tuples?

Les tuples regroupent plusieurs valeurs en une seule valeur composée. Les valeurs dans un tuple peuvent être de tout type et ne doivent pas nécessairement être du même type.

Les tuples sont créés en regroupant toute quantité de valeurs:

```
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
```

Des valeurs individuelles peuvent également être nommées lors de la définition du tuple:

```
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
```

Ils peuvent également être nommés lorsqu'ils sont utilisés en tant que variable et peuvent même avoir des valeurs facultatives à l'intérieur:

```
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
```

Se décomposer en variables individuelles

Les tuples peuvent être décomposés en variables individuelles avec la syntaxe suivante:

```
let myTuple = (name: "Some Name", age: 26)
let (first, second) = myTuple

print(first) // "Some Name"
print(second)  // 26
```

Cette syntaxe peut être utilisée indépendamment du fait que le tuple ait des propriétés non nommées:

```
let unnamedTuple = ("uno", "dos")
let (one, two) = unnamedTuple
print(one) // "uno"
print(two) // "dos"
```

Les propriétés spécifiques peuvent être ignorées en utilisant le trait de soulignement (_):

```
let longTuple = ("ichi", "ni", "san")
let (_, _, third) = longTuple
print(third) // "san"
```

Tuples comme valeur de retour des fonctions

Les fonctions peuvent renvoyer des tuples:

```
func tupleReturner() -> (Int, String) {
    return (3, "Hello")
}

let myTuple = tupleReturner()
print(myTuple.0) // 3
print(myTuple.1) // "Hello"
```

Si vous attribuez des noms de paramètres, ils peuvent être utilisés à partir de la valeur de retour:

```
func tupleReturner() -> (anInteger: Int, aString: String) {
    return (3, "Hello")
}

let myTuple = tupleReturner()
print(myTuple.anInteger) // 3
print(myTuple.aString) // "Hello"
```

Utiliser un typealias pour nommer votre type de tuple

Parfois, vous pouvez utiliser le même type de tuple à plusieurs endroits dans votre code. Cela peut rapidement devenir compliqué, surtout si votre tuple est complexe:

```
// 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 vous utilisez un certain type de tuple à plusieurs endroits, vous pouvez utiliser le mot-clé typealias pour nommer votre type de tuple.

```
// 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)
}
```

Si vous le faites trop souvent, vous devriez plutôt envisager d'utiliser une struct .

Echange de valeurs

Les tuples sont utiles pour échanger des valeurs entre 2 (ou plus) variables sans utiliser de variables temporaires.

Exemple avec 2 variables

Compte tenu de 2 variables

```
var a = "Marty McFly"
var b = "Emmett Brown"
```

nous pouvons facilement échanger les valeurs

```
(a, b) = (b, a)
```

Résultat:

```
print(a) // "Emmett Brown"
print(b) // "Marty McFly"
```

Exemple avec 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
```

Tuples comme cas dans Switch

Utiliser des tuples dans un commutateur

```
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
}
```

Ou en combinaison avec un Enum Par exemple avec Size Classes:

```
    let switchTuple = (UIUserInterfaceSizeClass.Compact, UIUserInterfaceSizeClass.Regular)

    switch switchTuple {
    case (.Regular, .Compact):
        //statement
    case (.Regular, .Regular):
        //statement
    case (.Compact, .Regular):
        //statement
    case (.Compact, .Compact):
        //statement
    }
```

Lire Tuples en ligne: https://riptutorial.com/fr/swift/topic/574/tuples

Chapitre 59: Typealias

Examples

typealias pour les fermetures avec paramètres

```
typealias SuccessHandler = (NSURLSessionDataTask, AnyObject?) -> Void
```

Ce bloc de code crée un alias de type nommé SuccessHandler , de la même manière que var string = "" crée une variable avec la string nom.

Maintenant, chaque fois que vous utilisez SuccessHandler , par exemple:

```
func example(_ handler: SuccessHandler) {}
```

Vous écrivez essentiellement:

```
func example(_ handler: (NSURLSessionDataTask, AnyObject?) -> Void) {}
```

typealias pour les fermetures vides

```
typealias Handler = () -> Void
typealias Handler = () -> ()
```

Ce bloc crée un alias de type qui fonctionne comme une fonction Void to Void (ne prend aucun paramètre et ne renvoie rien).

Voici un exemple d'utilisation:

```
var func: Handler?

func = {}
```

typealias pour d'autres types

```
typealias Number = NSNumber
```

Vous pouvez également utiliser un alias de type pour donner un autre nom à un type afin de le rendre plus facile à mémoriser ou pour rendre votre code plus élégant.

typealias pour Tuples

```
typealias PersonTuple = (name: String, age: Int, address: String)
```

Et cela peut être utilisé comme:

```
func getPerson(for name: String) -> PersonTuple {
    //fetch from db, etc
    return ("name", 45, "address")
}
```

Lire Typealias en ligne: https://riptutorial.com/fr/swift/topic/7552/typealias

Chapitre 60: Variables & Propriétés

Remarques

Propriétés : Associé à un type

Variables : non associées à un type

Consultez l' iBook du langage de programmation Swift pour plus d'informations.

Examples

Créer une variable

Déclarez une nouvelle variable avec var , suivie d'un nom, d'un type et d'une valeur:

```
var num: Int = 10
```

Les variables peuvent avoir leurs valeurs modifiées:

```
num = 20 // num now equals 20
```

Sauf s'ils sont définis avec let :

```
let num: Int = 10 // num cannot change
```

Swift déduit le type de variable, vous n'avez donc pas toujours à déclarer le type 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
```

Les noms de variables ne sont pas limités aux lettres et aux chiffres - ils peuvent également contenir la plupart des autres caractères Unicode, bien qu'il existe certaines restrictions.

> Les noms constants et variables ne peuvent pas contenir de caractères d'espacement, de symboles mathématiques, de flèches, de points de code Unicode à usage privé (ou non valides), ni de caractères de dessin de lignes et de boîtes. Ils ne peuvent pas non plus commencer par un chiffre
>
> **Source** developer.apple.com

```
var π: Double = 3.14159
var □□: String = "Apples"
```

Notions de base de la propriété

Les propriétés peuvent être ajoutées à une classe ou struct (techniquement énumérations aussi, voir exemple « Propriétés) calculée ». Celles-ci ajoutent des valeurs associées aux instances de classes / struct:

```
class Dog {
    var name = ""
}
```

Dans le cas ci-dessus, les instances de Dog ont une propriété nommée name de type String . La propriété est accessible et modifiée sur les instances de Dog :

```
let myDog = Dog()
myDog.name = "Doggy" // myDog's name is now "Doggy"
```

Ces types de propriétés sont considérés comme **des propriétés stockées** , car ils stockent quelque chose sur un objet et affectent sa mémoire.

Propriétés stockées paresseuses

Les propriétés stockées paresseuses ont des valeurs qui ne sont pas calculées avant leur première utilisation. Ceci est utile pour économiser de la mémoire lorsque le calcul de la variable est coûteux en calcul. Vous déclarez une propriété lazy avec lazy :

```
lazy var veryExpensiveVariable = expensiveMethod()
```

Souvent, il est affecté à une valeur de retour d'une fermeture:

```
lazy var veryExpensiveString = { () -> String in
    var str = expensiveStrFetch()
    str.expensiveManipulation(integer: arc4random_uniform(5))
    return str
}()
```

Les propriétés stockées paresseuses doivent être déclarées avec var .

Propriétés calculées

Différentes des propriétés stockées, **les propriétés calculées** sont construites avec un getter et un setter, exécutant le code nécessaire lorsque vous y accédez et définissez. Les propriétés calculées doivent définir un type:

```
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..."
```

Une propriété calculée en lecture seule est toujours déclarée avec un var :

```
var circumference: Double {
    get {
```

```
        return pi * radius * 2
    }
}
```

Les propriétés calculées en lecture seule peuvent être raccourcies pour exclure get :

```
var circumference: Double {
    return pi * radius * 2
}
```

Variables locales et globales

Les variables locales sont définies dans une fonction, une méthode ou une fermeture:

```
func printSomething() {
    let localString = "I'm local!"
    print(localString)
}

func printSomethingAgain() {
    print(localString) // error
}
```

Les variables globales sont définies en dehors d'une fonction, d'une méthode ou d'une fermeture et ne sont pas définies dans un type (pensez en dehors de tous les crochets). Ils peuvent être utilisés n'importe où:

```
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!
    }
}
```

Les variables globales sont définies paresseusement (voir exemple "Propriétés paresseuses").

Propriétés du type

Les propriétés de type sont des propriétés sur le type même, pas sur l'instance. Ils peuvent être à la fois des propriétés stockées ou calculées. Vous déclarez une propriété de type avec static :

```
struct Dog {
    static var noise = "Bark!"
}

print(Dog.noise) // Prints "Bark!"
```

Dans une classe, vous pouvez utiliser le mot class clé class au lieu de static pour le rendre remplaçable. Cependant, vous ne pouvez appliquer cela que sur les propriétés calculées:

```
class Animal {
    class var noise: String {
        return "Animal noise!"
    }
}
class Pig: Animal {
    override class var noise: String {
        return "Oink oink!"
    }
}
```

Ceci est souvent utilisé avec le modèle singleton .

Observateurs de propriété

Les observateurs de propriétés répondent aux modifications apportées à la valeur d'une propriété.

```
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 est appelé **avant que** myProperty soit défini. La nouvelle valeur est disponible en tant que newValue et l'ancienne valeur est toujours disponible en tant que myProperty .
- didSet est appelé **après que** myProperty soit défini. L'ancienne valeur est disponible sous la forme oldValue et la nouvelle valeur est désormais disponible sous la forme myProperty .

  Remarque: didSet et willSet ne seront pas appelés dans les cas suivants:

  - Assigner une valeur initiale
  - Modifier la variable dans son propre didSet ou willSet

- Les noms de paramètres pour oldValue et newValue de didSet et willSet peuvent également être déclarés pour améliorer la lisibilité:

```
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)")
    }
}
```

Attention: Bien qu'il soit possible de déclarer des noms de paramètres setter, il faut être prudent de ne pas mélanger les noms:

- willSet(oldValue) et didSet(newValue) sont entièrement légaux, mais vont

considérablement perturber les lecteurs de votre code.

Lire Variables & Propriétés en ligne: https://riptutorial.com/fr/swift/topic/536/variables--amp--proprietes

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec 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	(Unsafe) Buffer Pointers	Tommie C.
3	Algorithmes avec Swift	Austin Conlon, Bohdan Savych, Hady Nourallah, SteBra, Stephen Leppik, Tommie C.
4	Balisage de la documentation	Abdul Yasin, Martin Delille, Moritz, Rashwan L
5	Blocs	Matt
6	Booléens	Andreas, jtbandes, Kevin, pableiros
7	Boucles	Caleb Kleveter, D31, Efraim Weiss, Fred Faust, Hamish, Idan, Irfan, Jeff Lewis, Luca Angeletti, Moritz, Mr. Xcoder, Saagar Jha, Santa Claus, WMios, xoudini
8	Casting de type	Anand Nimje, andyvn22, godisgood4, LopSae, Nick Podratz
9	Chaînes et caractères	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
10	Commutateur	Ajwhiteway, AK1, Duncan C, elprl, Harshal Bhavsar, joan, Josh Brown, Luca Angeletti, Moritz, Santa Claus, ThrowingSpoon
11	Concurrence	Adda_25, Ahmad F, FelixSFD, JAL, LukeSideWalker, M_G, Matthew Seaman, Palle, Rob, Santa Claus
12	Conditionnels	AK1, atxe, Brduca, Community, Dalija Prasnikar, DarkDust, Hamish, jtbandes, ThaNerd, Thomas Gerot, tktsubota, toofani, torinpitchers
13	Contrôle d'accès	4444, Asdrubal, FelixSFD
14	Conventions de style	Grimxn, Moritz, Palle, Ryan H.
15	Cryptage AES	Matt, Stephen Leppik, zaph
16	Dérivation de clé PBKDF2	BUZZE, zaph

17	Des classes	Dalija Prasnikar, esthepiking, FelixSFD, jtbandes, Luca Angeletti, Matt, Ryan H., tktsubota, Tommie C., Zack
18	Design Patterns - Créatif	Ahmad F, AMAN77, Brduca, Dalija Prasnikar, Ian Rahman, Moritz, SeanRobinson159, SimpleBeat, Sơn Đỗ Đình Thy, Stephen Leppik, Thorax, Tommie C.
19	Design Patterns - Structural	Ian Rahman
20	Dictionnaires	dasdom, Diogo Antunes, egor.zhdan, iOSDevCenter, Jason Bourne, Kirit Modi, Koushik, Magisch, Moritz, RamenChef, Saagar Jha, sasquatch, Suneet Tipirneni, That lazy iOS Guy , ThrowingSpoon
21	Enregistrement dans Swift	Adam Bardon, D4ttatraya, DanHabib, jglasse, Moritz, paper1111, RamenChef
22	Ensembles	Community, Dalija Prasnikar, Luca Angeletti, Moritz, Steve Moser
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	Fermetures	ctietze, Duncan C, Hamish, Jojodmo, jtbandes, LopSae, Matthew Seaman, Moritz, Timothy Rascher, Tom Magnusson
25	Fonctionne comme des citoyens de première classe à Swift	Kumar Vivek Mitra
26	Fonctions Swift Advance	DarkDust, Sagar Thummar
27	Générer UIImage d'Initiales à partir de String	RubberDucky4444
28	Génériques	Andrey Gordeev, DarkDust, FelixSFD, Glenn R. Fisher, Hamish, Jojodmo, Kent Liau, Luca D'Alberti, Suneet Tipirneni, Ven, xoudini
29	Gestion de la mémoire	Accepted Answer, Daniel Firsht, jtbandes, Marc Gravell, Moritz, Palle, Tricertops
30	Gestionnaire d'achèvement	Maysam, Moritz
31	Gestionnaire de paquets rapide	Moritz
32	Hachage cryptographique	zaph
33	Initialiseurs	Brduca, FelixSFD, rashfmnb, Santa Claus, Vinupriya Arivazhagan
34	Injection de dépendance	Bear with me, JPetric
35	La déclaration	Palle

	différée	
36	La gestion des erreurs	Anil Varghese, cpimhoff, egor.zhdan, Jason Bourne, jtbandes, Mehul Sojitra, Moritz, Tom Magnusson
37	Lecture et écriture 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
38	Les extensions	Brduca, David, Esqarrouth, Jojodmo, jtbandes, Luca Angeletti, Moritz, rigdonmr
39	Les fonctions	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
40	Méthode Swizzling	JAL, Noam, Umberto Raimondi
41	Mise en cache sur l'espace disque	Viktor Gardart
42	Nombres	Arsen, jtbandes, Suragch, WMios, ZGski
43	NSRegularExpression dans Swift	Echelon, Hady Nourallah, ThrowingSpoon
44	Objets associés	Fattie, JAL
45	Opérateurs avancés	avismara, egor.zhdan, Fluidity, Hamish, Intentss, JAL, jtbandes , kennytm, Matthew Seaman, orccrusher99, tharkay
46	Options	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	OptionSet	4444, Alessandro
48	Performance	Matthew Seaman
49	Premiers pas avec la programmation orientée protocole	Alessandro Orrù, Fred Faust, kabiroberai, Krzysztof Romanowski
50	Programmation fonctionnelle dans Swift	Echelon, Luca Angeletti, Luke, Matthew Seaman, Shijing Lv
51	Protocoles	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
52	Réflexion	Asdrubal, LopSae, Sajjon
53	RxSwift	Alexander Olferuk, FelixSFD, imagngames, Moritz, Victor Sigler
54	Structs	Accepted Answer, AK1, Diogo Antunes, fredpi, Josh Brown, Kevin,

		Luca Angeletti, Marcus Rossel, Moritz, pbush25, Rob Napier, SamG
55	Swift HTTP server par Kitura	Fangming Ning
56	Tableaux	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
57	Travailler avec C et Objective-C	4444, Accepted Answer, jtbandes, Mark
58	Tuples	Accepted Answer, BaSha, Caleb Kleveter, JAL, Jason Sturges, Jojodmo, kabiroberai, LopSae, Luca Angeletti, Moritz, Nathan Kellert, Rick Pasveer, Ronald Martin, tktsubota
59	Typealias	Bartłomiej Semańczyk, Caleb Kleveter, D4ttatraya, Moritz
60	Variables & Propriétés	Christopher Oezbek, FelixSFD, Jojodmo, Luke, Santa Claus, tktsubota