Concept fondamentaux Swift

protocoles, extensions, et optionnelles

Antoine Moevus

Table des matières

Références

- https://www.hackingwithswift.com/example-code/language/what-are-indirect-enums
- Le livre de référence Swift

Vidéos explicatives

1. Introduction
2. Le secret des optionnels
3. Les protocoles
4. Les extensions
5. POP
6. Les fermetures et les énumérations avancées

Introduction

Les protocoles, les extensions, et les optionnelles sont trois concepts essentiels à la compréhension et à l'utilisation efficace du langage de programmation Swift.

Swift est un langage de programmation puissant et intuitif développé par Apple pour les systèmes iOS, macOS, watchOS et tvOS. Il est facile à apprendre, même pour ceux qui n'ont jamais codé auparavant, et possède une variété de caractéristiques qui le rendent flexible et puissant.

Dans ce cours, nous nous concentrerons sur trois de ces caractéristiques :

- Les **optionnelles**, qui permettent de gérer efficacement les valeurs potentiellement manquantes ou nulles.
- Les **protocoles**, qui définissent des modèles de méthodes, de propriétés et d'autres exigences pour une tâche ou une fonctionnalité particulière.
- Les **extensions**, qui permettent d'ajouter de nouvelles fonctionnalités à une classe, une structure ou une énumération existante.

En maîtrisant ces concepts, vous serez bien préparé à écrire du code Swift efficace et adaptable.

Optionnelles

Dans cette section, nous allons découvrir un concept fondamental de Swift : les optionnelles.

Définition et utilité des optionnelles

Les optionnelles sont un moyen que Swift utilise pour gérer l'absence de valeur, ou, en d'autres termes, pour gérer les situations où une variable pourrait être `null`. Une optionnelle signifie soit "il y a une valeur, et elle est égale à x" soit "il n'y a pas de valeur du tout".

```
var chaineOptionnelle: String? = "Bonjour"
chaineOptionnelle = nil // La variable peut être nulle
```

Optionnelles non nulles et nulles

Une optionnelle est soit nulle (`nil`), soit contient une valeur. Par exemple, une `String?` pourrait contenir une `String` ou être nulle. C'est ce que nous appelons une optionnelle non nulle ou nulle.

```
var chaineNonNulle: String? = "Bonjour"
var chaineNulle: String? = nil
```

Enchaînement d'optionnelles

L'enchaînement d'optionnelles est un moyen de demander plusieurs propriétés ou méthodes sur une optionnelle qui pourrait finalement aboutir à une valeur nulle.

```
var texte: String? = "Bonjour"
var longueurDuTexte = texte?.count // La valeur sera de type Int?
```

L'enchaînement d'optionnels est une alternative au déballage forcé, vous pouvez utiliser l'enchaînement d'optionnels pour accéder à une propriété d'une valeur optionnelle, et pour vérifier si cet accès à la propriété a réussi.

Par exemple voici comment, après avoir créer une nouvelle instance de Personne, accéder à sa propriété `nombreDePieces` de manière sécuritaire :

```
class Personne {
    var residence: Residence?
}

class Residence {
    var nombreDePieces = 1
}
```

```swift
let john = Personne()
if let nombreDePieces = john.residence?.nombreDePieces {
    print("La résidence de John a \(nombreDePieces) pièce(s).")
} else {
    print("Impossible de récupérer le nombre de pièces.")
}
// Affiche "Impossible de récupérer le nombre de pièces."
```

Coercition optionnelle et déballage forcé

La coercition optionnelle permet d'essayer de déballer une optionnelle et de retourner une valeur par défaut si elle est nulle. Le déballage forcé, représenté par un point d'exclamation (!) après l'optionnelle, est une façon d'extraire la valeur d'une optionnelle, mais cela provoquera une erreur si l'optionnelle est nulle.

```swift
var texte: String? = "Bonjour"
var longueurDuTexte = texte?.count ?? 0 // Coercition optionnelle
var texteForce = texte! // Déballage forcé
```

Protocoles en Swift

Les protocoles définissent des modèles de méthodes, de propriétés, et d'autres exigences pour contraindre un comportement particulier.

Définition et utilité des protocoles

Les protocoles sont un moyen de définir une interface de comportement que les autres types sont obligés de respecter. Dans Swift, vous pouvez utiliser les protocoles pour déclarer les méthodes et les propriétés qui doivent être implémentées par un type.

```
protocol Identifiable {
    var id: String { get set }
}
```

Dans cet exemple, `Identifiable` est un protocole qui exige une propriété `id` qui est lisible et écrivable.

Déclaration de protocoles

Pour déclarer un protocole, utilisez le mot-clé `protocol`. Vous pouvez spécifier les méthodes et les propriétés que le protocole exige, ainsi que leur visibilité (lecture seule ou lecture-écriture).

```
protocol PeutSeDeplacer {
    func deplacer()
}
```

Ici, le protocole `PeutSeDeplacer` définit une méthode `deplacer` que les types conformes doivent implémenter.

Conformité aux protocoles

Pour indiquer qu'un type doit se conformer à un certain protocole, vous utilisez le nom du protocole après le nom du type.

```
struct Point: Identifiable {
    var id: String
}
```

Dans cet exemple, `Point` est une structure qui se conforme au protocole `Identifiable` en fournissant une propriété `id`.

Héritage de protocole

Un protocole peut hériter d'un ou de plusieurs autres protocoles et peut ajouter de nouvelles exigences à ceux qu'il hérite.

```
protocol IdentifiableEtImprimable: Identifiable {
    func imprimer()
}
```

`IdentifiableEtImprimable` est un protocole qui hérite du protocole `Identifiable` et ajoute une exigence de méthode `imprimer()`.

Ceci dit, cela ne vient pas exploiter une particularité très utile des protocoles: l'héritage multiple. Ainsi le code suivant va être plus modulable et plus clair:

```
protocol Identifiable {
    var id: String { get set }
}

protocol Imprimable {
    func imprimer()
}

// L'héritage multiple entre les protocoles
protocol IdentifiableEtImprimable: Identifiable, Imprimable {
    // Vous pouvez ajouter d'autres exigences ici si nécessaire.
}

struct Objet: IdentifiableEtImprimable {
    var id: String

    func imprimer() {
        print("ID de l'objet: \(id)")
    }
}

let monObjet = Objet(id: "123")
monObjet.imprimer()  // Affiche "ID de l'objet: 123"
```

Note: Une classe, strutucture, ou énumération peut hériter de plusieurs protocoles aussi.

```
struct Objet: Imprimable, Identifiable {
    var id: String

    func imprimer() {
        print("ID de l'objet: \(id)")
    }
```

```
}
```

Résumé

Les protocoles en Swift sont un puissant outil de typage statique qui vous permet de définir des interfaces de comportement pour vos types. Ils permettent une forme de polymorphisme et d'abstraction qui peut aider à rendre votre code plus flexible et réutilisable. Leur utilisation peut grandement améliorer la lisibilité de votre code et faciliter sa maintenance.

Extensions en Swift

Un puissant outil qui permet de rajouter des fonctionnalités à des types existants.

Définition et utilité des extensions

Les extensions permettent d'ajouter de nouvelles fonctionnalités à une classe, une structure, une énumération ou un protocole existant. Cela comprend la capacité d'étendre des types pour lesquels vous n'avez pas accès au code source original (connu sous le nom de "types de bibliothèque"). Les extensions en Swift sont similaires aux catégories en Objective-C.

```
// Un exemple de définition d'extension sur le type Int pour ajouter une
fonctionnalité.
extension Int {
    func carre() -> Int {
        return self * self
    }
}
```

Déclaration d'extensions

Pour déclarer une extension, utilisez le mot-clé `extension` suivi du nom du type que vous voulez étendre. Vous pouvez ensuite ajouter de nouvelles fonctionnalités dans le bloc de l'extension.

```
extension String {
    func saluer() -> String {
        return "Bonjour, \(self)!"
    }
}
```

Ajout de méthodes d'instance

Avec les extensions, vous pouvez ajouter des méthodes d'instance à un type existant. Cela permet de personnaliser et d'étendre les fonctionnalités du type.

```
extension Int {
    func estPair() -> Bool {
        return self % 2 == 0
    }
}
```

Exemple d’application à la classe `String`

En utilisant une extension, nous pouvons, par exemple, ajouter une méthode à la classe `String` de Swift pour créer une salutation personnalisée.

```
extension String {
    func saluer() -> String {
        return "Bonjour, \(self)!"
    }
}

let salutation = "Paul".saluer()
print(salutation) // Affiche "Bonjour, Paul!"
```

Programmation Orientée Protocole (POP)

La Programmation Orientée Protocole est un paradigme de programmation puissant et flexible popularisé par Swift. Elle utilise des protocoles comme base pour la création de comportements et d'interactions modulaires et réutilisables.

Définition et utilité de la POP

La Programmation Orientée Protocole met l'accent sur l'utilisation de protocoles pour définir des contrats de comportement et d'interaction entre les entités de votre code. C'est un moyen efficace de partager du code et de créer des modèles de conception flexibles. Il existe plusieurs techniques propre à Swift afin de programmer en utilisant le plein potentiel des protocoles. Le premier mécanisme est l'héritage multiple (aussi parfois appellé composition de protocole). Les principaux autres sont l'exigence de mise en oeuvre, et l'implémentation par défaut.

Protocoles avec exigence de mise en œuvre

Avec Swift, vous pouvez définir des protocoles qui requièrent des types conformes pour implémenter certaines méthodes ou propriétés.

```
protocol PeutSeDeplacer {
    func deplacer()
}
```

Protocoles avec des implémentations par défaut

Swift permet également aux protocoles de fournir une implémentation par défaut pour certaines méthodes ou propriétés. Cela permet de partager du code tout en offrant la possibilité de le personnaliser pour des types spécifiques.

```
protocol PeutSeDeplacer {
    func deplacer()
}

extension PeutSeDeplacer {
    func deplacer() {
        print("Se déplace à une vitesse standard.")
    }
}
```

Exemple récapitulatif

```
// Définition d'un protocole "Volant"
protocol Volant {
```

```
    var altitudeMaximale: Int { get }
    func voler() -> String
}

// Définition d'un protocole "ConsommeEssence"
protocol ConsommeEssence {
    var consommationEssenceParKilometre: Double { get }
    func consommerEssence(distance: Double) -> String
}

// Extension sur le protocole "Volant" pour ajouter une méthode par défaut
extension Volant {
    func atteindreAltitudeMax() -> String {
        return "Atteindre une altitude de \(altitudeMaximale) mètres."
    }
}

// Classe "Oiseau" qui se conforme au protocole "Volant"
class Oiseau: Volant {
    var altitudeMaximale: Int

    init(altitudeMaximale: Int) {
        self.altitudeMaximale = altitudeMaximale
    }

    func voler() -> String {
        return "L'oiseau vole jusqu'à \(altitudeMaximale) mètres."
    }
}

// Classe "Avion" qui se conforme aux protocoles "Volant" et "ConsommeEssence"
class Avion: Volant, ConsommeEssence {
    var altitudeMaximale: Int
    var consommationEssenceParKilometre: Double

    init(altitudeMaximale: Int, consommationEssenceParKilometre: Double) {
        self.altitudeMaximale = altitudeMaximale
        self.consommationEssenceParKilometre = consommationEssenceParKilometre
    }

    func voler() -> String {
        return "L'avion vole jusqu'à \(altitudeMaximale) mètres."
    }

    func consommerEssence(distance: Double) -> String {
        let essenceConsommée = distance * consommationEssenceParKilometre
        return "L'avion a consommé \(essenceConsommée) litres d'essence pour parcourir
\(distance) kilomètres."
    }
}
```

```swift
// Création d'un objet "Avion"
let monAvion = Avion(altitudeMaximale: 10000, consommationEssenceParKilometre: 0.05)

// Appel de la méthode "consommerEssence()"
print(monAvion.consommerEssence(distance: 1000))


// Création d'un tableau d'objets "Volant"
let objetsVolants: [Volant] = [Oiseau(altitudeMaximale: 1000), Avion(altitudeMaximale:
10000)]

// Utilisation d'une closure pour filtrer les objets "Volant" qui peuvent voler au-
dessus de 5000 mètres
let objetsHauteAltitude = objetsVolants.filter { $0.altitudeMaximale > 5000 }

// Affichage des objets qui peuvent voler à haute altitude
for objet in objetsHauteAltitude {
    print(objet.voler())
}
```

Fermetures (Closures)

Les fermetures sont des blocs de fonctionnalités autonomes que vous pouvez passer et utiliser dans votre code.

Définition et utilité des fermetures

Les fermetures, également connues sous le nom de closures en anglais, sont des blocs de code qui peuvent être appelés ailleurs dans votre code. Ils peuvent capturer et stocker des références à des variables et des constantes du contexte dans lequel ils ont été définis. Cela leur permet de gérer ces variables plus tard, même si le contexte original n'existe plus.

```
let incrementer: () -> Int = {
    var total = 0
    return {
        total += 1
        return total
    }
}()
```

Syntaxe des fermetures

En Swift, la syntaxe des fermetures est flexible et peut être adaptée à différents besoins. Vous pouvez rendre une fermeture plus concise en omettant certains éléments, comme les types de retour, si Swift peut les inférer automatiquement.

```
let ajouter = { (x: Int, y: Int) -> Int in
    return x + y
}
let resultat = ajouter(1, 2)
```

Captures de valeurs

Les fermetures peuvent capturer des valeurs du contexte dans lequel elles sont définies. Cela signifie qu'elles peuvent accéder à ces valeurs plus tard, même si elles n'existent plus dans leur contexte original.

```
func faireIncrementer() -> () -> Int {
    var total = 0
    let incrementer: () -> Int = {
        total += 1
        return total
    }
    return incrementer
}
```

```
let incrementer = faireIncrementer()
incrementer() // renvoie 1
incrementer() // renvoie 2
```

Fermetures comme paramètres de fonction

Les fermetures peuvent être utilisées comme paramètres de fonction, ce qui vous permet de passer des blocs de code à vos fonctions.

```
func appliquerOperation(_ a: Int, _ b: Int, _ operation: (Int, Int) -> Int) -> Int {
    return operation(a, b)
}
let resultat = appliquerOperation(2, 3, ajouter)
```

Syntaxe abrégée des fermetures comme paramètres de fonction

Swift offre plusieurs façons de raccourcir une fermeture lorsque celle-ci est passée en tant que paramètre à une fonction. Cela se fait souvent dans le cas de méthodes de collections telles que `map`, `filter` et `reduce`, ou bien lorsque l'on implémentre le contraolleur du modèle MVC. Commençons par une fermeture complète et raccourcissons-la progressivement:

```
let nombres = [1, 2, 3, 4, 5]

// Complète Closure Syntax
let nombresDoubles = nombres.map({ (nombre: Int) -> Int in
    return nombre * 2
})

// Le contexte permet à Swift d'inférer les types de paramètres et de retour.
let nombresDoubles = nombres.map({ nombre in
    return nombre * 2
})

// Les fermetures à expression unique retournent implicitement le résultat, donc
'return' n'est pas nécessaire.
let nombresDoubles = nombres.map({ nombre in
    nombre * 2
})

// Swift fournit automatiquement les noms de paramètres courts et anonymes.
let nombresDoubles = nombres.map({
    $0 * 2
})

// Si la fermeture est le dernier argument, vous pouvez écrire une fermeture de style
"trailing" en dehors des parenthèses.
let nombresDoubles = nombres.map { $0 * 2 }
```

```
// Le résultat est [2, 4, 6, 8, 10]
print(nombresDoubles)
```

Voici d'autres exemples:

```
let nombres = [1, 2, 3, 4, 5]

// Utilisation de la fonction filter avec une closure
let nombresPairs = nombres.filter { $0 % 2 == 0 }
print(nombresPairs) // Résultat : [2, 4]

// Utilisation de la fonction reduce avec une closure
let somme = nombres.reduce(0, { $0 + $1 * 2 })
print(somme) // Résultat : 30

// Utilisation de la fonction sorted avec une closure
let mots = ["Swift", "Programming", "Language"]
let motsTries = mots.sorted { $0.count < $1.count }
print(motsTries) // Résultat : ["Swift", "Language", "Programming"]
```

Les énumérations avancées

Propriétés calculées

En Swift, vous pouvez ajouter des propriétés calculées à une énumération pour fournir des valeurs supplémentaires qui sont déterminées à partir de l'état de l'énumération.

```
enum Direction {
    case nord, sud, est, ouest

    var description: String {
        switch self {
        case .nord:
            return "Nord"
        case .sud:
            return "Sud"
        case .est:
            return "Est"
        case .ouest:
            return "Ouest"
        }
    }
}
```

Dans cet exemple, `Direction` est une énumération qui a une propriété calculée `description`. Cette propriété retourne une chaîne de caractères qui décrit la direction.

Vous pouvez accéder à une propriété calculée sur une instance d'une énumération de la même manière que vous accéderiez à une propriété normale.

```
let maDirection = Direction.est
print(maDirection.description) // affiche "Est"
```

Énumérations récursives

Les énumérations récursives (ou indirectes) sont une fonctionnalité puissante de Swift qui vous permet de créer une énumération dont l'un des cas peut avoir une instance de l'énumération elle-même comme valeur associée. Cela permet de créer des structures de données complexes, comme des listes liées ou des arbres.

Pour déclarer une énumération récursive, vous devez marquer l'énumération avec le mot-clé `indirect`, soit devant l'énumération elle-même, soit devant le cas qui fait référence à l'énumération.

```
indirect enum ListeLien<T> {
    case fin
    case noeud(T, suivant: ListeLien<T>)
```

```
}
```

Dans cet exemple, `ListeLien<T>` est une énumération générique qui définit une liste liée. L'énumération est marquée avec `indirect` pour indiquer qu'elle est récursive : le cas `noeud` a une valeur associée qui est une autre instance de `ListeLien<T>`.

Vous pouvez utiliser cette énumération pour créer une liste liée d'entiers comme suit :

```
let maListe = ListeLien.noeud(1, suivant: .noeud(2, suivant: .noeud(3, suivant: .
fin)))
```

Exemple : Pièces de Voiture

Prenons l'exemple d'une voiture qui peut être décomposée en différentes pièces principales, chacune pouvant elle-même être décomposée en sous-pièces.

```
indirect enum PieceVoiture {
    case voiture(String, [PieceVoiture])
    case moteur(String, [PieceVoiture])
    case roue(String)
}
```

Dans cet exemple, une voiture est composée de différentes pièces, dont un moteur et des roues. Le moteur peut également être décomposé en sous-pièces.

```
let maVoiture = PieceVoiture.voiture("MaVoiture", [
    .moteur("MoteurPrincipal", []),
    .roue("RoueAvantGauche"),
    .roue("RoueAvantDroite"),
    .roue("RoueArriereGauche"),
    .roue("RoueArriereDroite")
])
```

Exemple : Modes d'affichage

```
indirect enum ModeAffichage {
    case mobile
    case ipad
    case block
    case inline
    case multi(String, [ModeAffichage])
}

let modeAffichage = ModeAffichage.collection("Barre de Naviguation", [.inline, .ipad])
```

Les différents cas de l'énumération sont `mobile`, `ipad`, `block`, `inline` et `multi`. Chaque cas représente un mode d'affichage différent. Par exemple, `mobile` pourrait être utilisé pour configurer l'interface utilisateur pour les petits écrans de téléphone, tandis que `ipad` pourrait être utilisé pour les écrans plus grands.

Cette énumération est `indirect`, ce qui signifie qu'elle peut être utilisée de manière récursive. C'est ce qui permet le cas `multi`, qui a une valeur associée qui est une chaîne (un nom descriptif ou un titre) et un tableau de `ModeAffichage`. Cela permet de regrouper plusieurs modes d'affichage dans une seule unité logique.

Ensuite, une variable `modeAffichage` est définie comme une instance de `ModeAffichage.collection`. C'est une "collection" nommée "Barre de Navigation" qui regroupe les modes `inline` et `ipad`.

L'intérêt de ce code est qu'il fournit une structure flexible pour représenter et manipuler les modes d'affichage dans une application. Cela pourrait être utilisé, par exemple, pour changer dynamiquement l'apparence de l'application en fonction du mode d'affichage actuel.

Excercices

Enchaînement d'optionnelles

1. Créez un objet de classe avec une propriété optionnelle, puis essayez d'accéder à cette propriété en utilisant l'enchaînement optionnel.
2. Ajoutez une propriété optionnelle à une structure, puis essayez d'accéder à cette propriété en utilisant l'enchaînement optionnel.
3. Créez un dictionnaire avec des clés de type `String` et des valeurs optionnelles de type `Int`. Essayez d'accéder à une valeur en utilisant l'enchaînement optionnel.

Coercition optionnelle et déballage forcé

1. Créez une variable optionnelle, puis utilisez la coercition pour assigner une valeur par défaut si l'optionnelle est `nil`.
2. Créez une variable optionnelle, puis utilisez le déballage forcé pour accéder à sa valeur. Que se passe-t-il si l'optionnelle est `nil` ?
3. Créez une fonction qui accepte une variable optionnelle en tant que paramètre, puis utilisez la coercition optionnelle pour assigner une valeur par défaut si l'optionnelle est `nil` dans le corps de la fonction.

Protocoles

1. Créez un protocole `Animal` qui spécifie une propriété `nom` et une méthode `faireDuBruit()`.
2. Créez des structures `Chien` et `Chat` qui se conforment au protocole `Animal`. Créez un tableau de cinq instances d'`Animal`
3. Définissez un protocole `Identifiable` avec une propriété `id` de type `String` et implémentez-le dans une structure `Personne`.
4. Créez un protocole `Comparable` avec une méthode `compareTo(other:) `qui prend un autre objet du même type et renvoie un `Bool`. Implémentez ce protocole dans une structure `Chiffre`.
5. Créez un protocole `Imprimable` avec une méthode `imprimer()`. Faites en sorte qu'une structure `Document` se conforme à ce protocole.

Extensions

1. Créez une extension pour le type `Double` qui ajoute une méthode `carre()` pour calculer le carré du nombre.
2. Créez une extension pour le type `String` qui ajoute une propriété calculée `estPalindrome` pour vérifier si une chaîne est un palindrome.
3. Créez une extension pour le type `Array` où les éléments sont `Comparable`. Ajoutez une méthode `max()` qui renvoie l'élément maximum du tableau.
4. Définissez un protocole `Reproductible` qui exige une méthode `dupliquer()`. Créez une extension

de `Array` où les éléments sont `Reproductible` qui ajoute une méthode `dupliquerTous()` qui renvoie un nouveau tableau avec tous les éléments dupliqués.

Enumerations

1. Créez une énumération `JourDeLaSemaine` qui représente les jours de la semaine. Ajoutez une propriété calculée `estWeekend` qui retourne `true` si le jour est un samedi ou un dimanche, et `false` sinon.
2. Créez une énumération `Mois` qui représente les mois de l'année. Ajoutez une propriété calculée `nombreDeJours` qui retourne le nombre de jours dans le mois (ignorez les années bissextiles pour cet exercice).
3. Créez une énumération indirecte `ArbreBinaire` pour un arbre binaire contenant des entiers.
4. Créez une énumération indirecte `Liste` pour une liste chaînée de `String`.

Fermetures

1. Créez une fermeture qui prend deux `Int` en paramètres et retourne leur somme. Utilisez cette fermeture pour additionner deux nombres.
2. Créez une fonction qui prend une fermeture en tant que paramètre. Cette fermeture devrait prendre un `Int` et retourner un `Bool`. Utilisez cette fonction pour filtrer un tableau d'``Int``.
3. Créez une fermeture qui capture et modifie une valeur d'une variable extérieure.
4. Utilisez la syntaxe de fermeture simplifiée pour transformer un tableau de `String` en un tableau de leurs longueurs.
5. Créez une fermeture qui retourne une fonction. Cette dernière doit prendre un `Int` en paramètre et le multiplier par une valeur capturée par la fermeture.
6. Créez une fermeture qui agit comme une fonction de rappel (callback). Utilisez-la dans une fonction qui simule un téléchargement asynchrone.