Tutoriel 1 : Protocoles

Attention ces notions ne seront revues qu'au cours du prochain TP (aucun CM ne sera consacré à ces notions). Si vous avez des questions ou des difficultés au cours de ce tutoriel, notez les, un créneau sera réservé aux questions lors du prochain TP. Par ailleurs, un QCM sera organisé au début du prochain TP pour évaluer ce que vous avez retenu de ce tutoriel.

Dans la première partie de ce tutoriel, la notion de protocole vous sera présentée à travers un exemple simple. Dans la seconde partie, vous reproduirez la mise en place d'un objet de type `UITableView` qui se conforme à deux protocoles appelés `UITableViewDataSource` et `UITableViewDelegate`. Enfin, dans une troisième partie, vous créerez une application présentant à l'utilisateur un objet de type UIPickerView qui se conforme à deux protocoles appelés `UIPickerViewDataSource` et `UIPickerViewDelegate`.

Partie I : Définition d'un protocole

Selon le « Swift Programming Language Guide » rédigé par Apple, Un protocole définit un plan de méthodes, de propriétés et d'autres exigences qui conviennent à une tâche ou une fonctionnalité particulière. Le protocole peut ensuite être adopté par une classe, une structure ou une énumération pour fournir une implémentation réelle de ces exigences. Tout type qui répond aux exigences d'un protocole est dit conforme à ce protocole.

Ainsi, selon les créateurs du langage Swift, les protocoles sont un bon moyen de définir un ensemble de fonctionnalités requises que d'autres types peuvent adopter. Cela revient à dire que les protocoles fournissent des informations sur ce qu'un type peut faire, pas nécessairement ce qu'il est. Les classes et les structures vous fournissent des informations sur ce qu'est un objet, mais les protocoles vous fournissent des informations sur ce que fait un objet.

Protocoles vs. Sous-classement

L'un des modèles de conception les plus courants dans la programmation orientée objet est appelé sous-classement (subclassing en anglais). Le sous-classement vous permet de définir des relations parent / enfant entre les classes:

```
class MyClass { }
class MySubclass: MyClass { }
```

Dans la relation ci-dessus, `MySubclass` est une classe enfant (une sous-classe) de `MyClass`. `MySubclass` héritera automatiquement de toutes les fonctionnalités de `MyClass`, y compris les propriétés, les fonctions, les constructeurs, etc., ce qui signifie que tous les membres de `MyClass` seront automatiquement disponibles dans `MySubclass`.

```
class MyClass {
    var myProperty: String {
        return "property"
    }

    func myFunction() -> String {
        return "function"
    }

    init(string: String) {
        print("Initializing an instance of MyClass with \(string)!")
    }
}

class MySubclass: MyClass() { }

let s = MySubclass(string: "Hello world") //prints "Initializing an
instance of MyClass with hello, world!"
print(s.myProperty) //prints "property"
print(s.myFunction()) //prints "function"
```

Les sous-classes peuvent également remplacer (override) la fonctionnalité de leur superclasse, ce qui signifie qu'elles peuvent la remplacer par la leur:

```
class MySubclass: MyClass() {
    override var myProperty: String {
        return "overriden property"
    }

    override func myFunction() -> String {
        return "overriden function"
    }

    override init(string: String) {
        print("Initializing an instance of MySubclass with \(string)!")
    }
}

let s = MySubclass(string: "Hello world") //prints "Initializing an
instance of MySubclass with hello, world!"
print(s.myProperty) //prints "overriden property"
print(s.myFunction()) //prints "overridden function"
```

Le sous-classement est un modèle de conception très puissant car il permet aux développeurs de créer des relations incroyablement détaillées et riches entre les classes. Mais, aussi puissant qu'il soit, le sous-classement ne résout pas tous les problèmes que nous rencontrons lors de la création d'applications. Par exemple :

```
class Animal {
    func makeSound() { }
}
```

Nous venons de définir une classe, `Animal`, pour représenter différents types d'animaux au sein de notre application. Nous avons inclus une fonction, `makeSound()`, pour représenter la fonctionnalité que nos animaux peuvent posséder. Cependant, il y a un problème. Les animaux n'ont pas un cri unique, alors que pouvons-nous mettre dans la fonction `makeSound()` ? En pratique, nous pourrions faire quelque chose comme ceci:

```
class Animal {
    func makeSound() { fatalError("Implement me!") }
}
```

Ce code ajoute une erreur `fatalError` à la fonction définie dans `Animal`, faisant ainsi d'`Animal` une classe de base abstraite, ou une classe qui n'est pas instanciée directement. Les classes de base abstraites ne peuvent être instanciées qu'à travers leurs sous-classes. Maintenant, nous pouvons sous-classer `Animal` et définir nos propres animaux:

```
class Dog: Animal {
    override func makeSound() { print("Woof!") }
}
```

Nous avons désormais une classe `Dog` que nous pouvons instancier de la façon suivante :

```
let rex = Dog()
rex.makeSound() //prints "Woof!"
```

Que se passe-t-il si nous oublions de surcharger `makeSound()` ? Ou si nous essayons d'instancier directement un animal ?

```
let tim = Animal()
tim.makeSound() //CRASH
```

```
class Cat: Animal { }
let ginger = Cat()
ginger.makeSound() //CRASH
```

C'est donc une situation où le sous-classement n'est pas idéal. Nous voyons des situations comme celle-ci tout le temps. En substance, le sous-classement échoue chaque fois qu'il n'y a pas d'implémentation par défaut à utiliser dans une super-classe.
Revenons sur notre exemple animal, mais avec des protocoles:

```
protocol Sound {
            func makeSound()
}
```

Nous avons défini un protocole, `Sound`, qui spécifie que quelque chose doit avoir une fonction `makeSound()`. Peu importe ce que l'objet est vraiment, car nous nous soucions seulement de savoir s'il répond ou non à nos exigences pour le protocole `Sound`.

En pratique, cela donne le code suivant :

```
struct Dog: Sound {
    func makeSound() {
        print("Woof")
    }
}

struct Tree: Sound {
    func makeSound() {
        print("Susurrate")
    }
}

struct iPhone: Sound {
    func makeSound() {
        print("Ring")
    }
}
```

Ici, `Sound` est un protocole, nous pouvons donc étendre n'importe quel type pour qu'il se conforme à `Sound`, comme nous l'avons fait dans l'exemple ci-dessus. S'il est vrai que les chiens sont des animaux, les arbres et les iPhones ne le sont pas. Cela n'a donc pas de sens de les sous-classer à `Animal`. Dans le cas du protocole `Sound`, cela n'a pas d'importance. Nous avons décidé que la seule exigence pour qu'un objet puisse émettre un son est d'adopter le protocole et de mettre en œuvre la méthode requise.

Bilan

Un protocole sert à définir un plan minimal que doit suivre une classe. C'est en quelque sorte un modèle pour votre classe. La particularité ici est qu'elle ne fournit pas l'implémentation de vos méthodes (le contenu). On écrit juste les méthodes sans décrire ce qu'elles vont réellement faire au final. Dans un protocole, cela n'est pas notre but.

Tout seul, un protocole ne sert strictement à rien. En effet, chaque protocole est destiné à être implémenté par une classe qui devra implémenter les méthodes du protocole. Et cette fois-ci, le contenu de vos méthodes seront écrites. Un protocole ne doit pas être confondu avec l'héritage. Un chien est un animal. Une voiture est un véhicule. Mais un animal n'est pas un véhicule. Pourtant, les deux peuvent se déplacer : c'est là qu'interviennent les protocoles.

Pour une classe qui a implémenté un protocole, on dit que la classe est conforme au protocole.

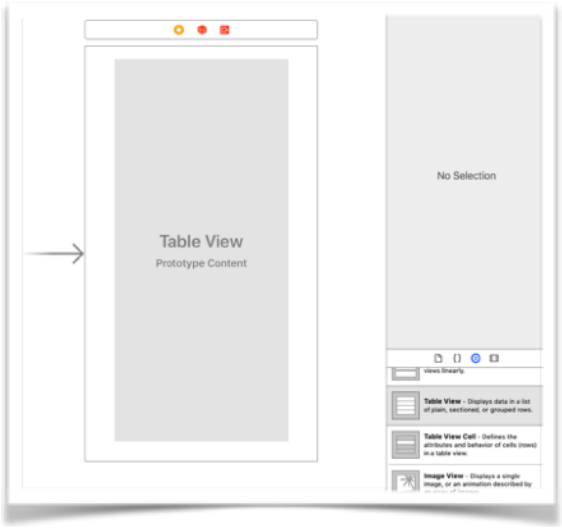
Pour ceux ayant des bases en programmation orientée objet, un protocole est exactement la même chose qu'une interface.

Partie II : Exemple d'utilisation de protocoles

Dans cette seconde partie vous allez suivre pas à pas le tutoriel proposé permettant implémenter un objet de type `UITableView` dans une application iOS. La classe `UITableView` peut se conformer à deux protocoles : le protocole `UITableViewDelegate` et le protocole `UITableViewDatasource`.

Sous Xcode, créez une nouvelle application de type Single View App.

Ajoutez un objet de type Table View (disponible dans la bibliothèque d'objets) à votre storyboard et créez un `IBOutlet` grâce à l'assistant d'édition pour associer une variable (`table`) à votre objet.



Ajouter à votre code une liste de chaîne de caractères `data` qui contient les données que vous souhaitez afficher dans votre objet de type `UITableView` :

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var table: UITableView!
    let data : [String] = ["Michel", "Arthur", "Jean", "Pierre",
"Yannick", "Morgan", "Pierre", "Baptiste", "Bruno", "Alban", "Oscar",
"Maxence"]
    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically
from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

Ajouter également les deux lignes suivantes dans la méthode `viewDidLoad()`

```
table.dataSource = self
table.delegate = self
```

Ces deux méthodes précisent que les méthodes des deux protocoles `UITableViewDelegate` et `UITableViewDataSource` vont être implémentées dans la classe courante.

Une fois ces deux lignes ajoutées, vous allez voir apparaitre les erreurs suivantes : « `Cannot assign value of type 'ViewController' to type 'UITableViewDataSource?` » et « `Cannot assign value of type 'ViewController' to type 'UITableViewDelegate?` ». Ces deux erreurs signifient qu'il faut que la classe `ViewController` se conforme aux protocoles que l'on essaie d'assigner sur ces deux lignes.

Pour préciser que notre classe `ViewController` se conforme à ces deux protocoles, il suffit de modifier l'en-tête de notre classe de la façon suivante :

```
class ViewController: UIViewController, UITableViewDelegate,
                                        UITableViewDataSource {
```

La modification de cette ligne, créée une nouvelle erreur : « `Type 'ViewController' does not conform to protocol 'UITableViewDataSource'` » car le protocole `UITableViewDataSource` contient des méthodes obligatoires que nous n'avons pas encore implémentées (le compilateur se plaint donc que notre classe ne se conforme pas vraiment au protocole, contrairement à ce que nous avons indiqué). Pour corriger cette erreur nous allons donc devoir implémenter les méthodes des protocoles.

Les protocoles `UITableViewDataSource` et `UITableViewDelegate`

Le protocole `UITableViewDataSource` est adopté par un objet qui sert de médiateur au modèle de données de l'application pour un objet `UITableView`. Le DataSource fournit à l'objet de type `UITableView` les informations dont il a besoin pour construire et modifier une TableView (https://developer.apple.com/documentation/uikit/uitableviewdatasource)

La documentation d'Apple indique que le protocole `UITableViewDataSource` contient deux méthodes qu'il est obligatoire d'implémenter :

- `tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int` : cette méthode doit retourner le nombre de lignes dans la section en paramètre de la tableview.
- `tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell` : cette méthode doit retourner la cellule correspondant à l'`IndexPath` donnée en paramètre (un `IndexPath` est un couple contenant le nombre de la section et le nombre de la ligne).

Le protocole `UITableViewDelegate` défini les méthodes à envoyer au délégué d'un objet `UITableView` : (https://developer.apple.com/documentation/uikit/uitableviewdelegate).

Les méthodes du protocole permettent au délégué de gérer les sélections, de configurer les en-têtes et les pieds de page, d'aider à supprimer et réorganiser les cellules et d'effectuer d'autres actions. Toutes les méthodes du protocole sont optionnelles, par exemple:

- `tableView(UITableView, didSelectRowAt: IndexPath) :` cette méthode indique au délégué que la ligne précisée par l'`IndexPath` a été sélectionnée par l'utilisateur

Mais qu'est ce qu'un délégué ?

Les délégués sont un modèle de conception qui permet à un objet d'envoyer des messages à un autre objet lorsqu'un événement spécifique se produit.

Imaginez qu'un objet A appelle un objet B pour effectuer une action. Une fois l'action terminée, l'objet A doit savoir que B a terminé la tâche et prendre les mesures nécessaires, ceci peut être réalisé avec l'aide des délégués!

Dans le cas des objets de type `UITableView`, votre objet `table` va envoyer des messages à son délégué lorsque certains évènements vont se produire (par exemple, si l'utilisateur sélectionne une ligne) afin que le protocole `UITableViewDelegate` sache quand ses méthodes doivent être appelées.

Implémentation des méthodes des protocoles

La première méthode obligatoire du protocole `UITableViewDataSource` que nous devons implémenter est la méthode `tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int` qui indique le nombre de lignes à afficher dans les sections de notre `UITableView` (dans cet exemple, notre `UITableView` ne possède qu'une seule section mais il est possible de créer plusieurs section dans une `UITableView`).

Le nombre de lignes à afficher correspond au nombre d'éléments contenus dans la liste de données `data` :

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section:
                                                              Int) -> Int {
    return data.count
}
```

La deuxième méthode obligatoire du protocole `UITableViewDataSource` retourne la cellule correspondant à l'`IndexPath` donnée en paramètre. Elle permet d'afficher le contenu de notre tableau `data` chaque ligne de notre objet `UITableView`.

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath:
                                            IndexPath) -> UITableViewCell {
    // 1. Récupération du bon élément dans notre tableau data
    let d = data[indexPath.row]

    // 2. Utilisation d'une cellule existante ou création d'une
nouvelle cellule
    let cellIdentifier = "ElementCell"
     let cell = tableView.dequeueReusableCell(withIdentifier:
                                                    cellIdentifier)
        ?? UITableViewCell(style: .subtitle, reuseIdentifier:
                                                    cellIdentifier)

    // 3. Ajout des informations dans le champ texte de cellule
    cell.textLabel?.text = d

    // 4. Renvoie de la cellule
    return cell
}
```

La deuxième étape de la fonction permet une meilleure gestion de la mémoire de l'application en créant une cellule réutilisable pour afficher la première ligne puis en réutilisant cette cellule pour les suivantes (plutôt que de créer un nouvelle objet de type cellule pour chaque ligne).

Finalement, il est possible d'implémenter une méthode optionnelle du protocole afin d'afficher dans la console le contenu de la ligne de notre objet de type `UITableView` sélectionnée par l'utilisateur.

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
                                                        IndexPath) {
    print(data[indexPath.row])
}
```

Testez votre application.

Partie III : Implémentation d'un UIPickerView

Dans cette troisième partie vous allez implémenter un objet de type `UIPickerView` dans une application iOS. La classe `UIPickerView` peut se conformer à deux protocoles : le protocole `UIPickerViewDelegate` et le protocole `UIPickerViewDatasource`.

A la fin de cet exercice votre application doit ressembler à ça :

Sous Xcode, créez une nouvelle application de type Single View App.

Ajoutez un objet de type `UIPickerView` (disponible dans la bibliothèque d'objets) à votre storyboard et créez un `IBOutlet` grâce à l'assistant d'édition pour associer une variable à votre objet.

Créez trois structures de données qui vont contenir les données à afficher dans le `UIPickerView`.

```
let data1 : [String] = ["Café", "Thé", "Cappuccino", "Tisane"]
let data2 : [String] = ["chaud", "tiède", "froid"]
let data3 : [String] = ["sucré", "faiblement sucré", "non sucré"]
```

Créez trois variables qui vont permettre de stocker le choix de l'utilisateur

```
var boisson : String = ""
var chaleur : String = ""
var sucre : String = ""
```

Implémentez les méthodes du protocole `UIPickerViewDataSource` suivantes :

```
func numberOfComponents(in pickerView: UIPickerView) -> Int
func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int
```

Plus d'information sur ce protocole ici : https://developer.apple.com/documentation/uikit/uipickerviewdatasource

Conseil : votre `UIPickerView` va posséder 3 composants (pour afficher les trois tableaux de données). Vous pouvez faire référence à chacun de ces composants grâce à leurs indices (0, 1 ou 2) par le biais du paramètre `component`.

Implémentez les méthodes du protocole `UIPickerViewDataSource` suivantes :

```
func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String?
func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent component: Int)
```

Plus d'information sur ce protocole ici : https://developer.apple.com/documentation/uikit/uipickerviewdelegate

Finalement, implémentez la méthode liée au bouton commander afin de faire apparaitre la commande dans le label de votre interface graphique.

Déposez votre application sur Moodle.

Sources :

- https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Protocols.html
- https://www.appcoda.com/protocols-in-swift/
- https://blog.thomasdurand.fr/swift3/ios/2016/08/07/bases-ios-configuration-tableview.html