

DEVELOPPEMENT MOBILE iOS avec Swift

Programmation Orient é Objet

INSA BADJI Doctorant à l'Université de Thiès / Tuteur à



Séquence 2: Swift

- Plan
    - Programmation orientée objet:
        - Objets
        - Classes
        - Enum
        - Strunc

Programmation Orienté Object (POO)

- Swift est un langage hérité de C et Objective-C, il est à la fois un langage procédural, à la fois comme langue orientée objet. La classe (Class) est un concept du langage de programmation orienté objet. La classe a des propriétés et des méthodes, naturellement, la méthode (method) est comprise comme une fonction de la classe. De la classe, vous pouvez créer des objets.
- Swift utilise le mot-clé de la class pour déclarer une classe.
- Un objet permet de :
  - cacher une implémentation
  - représenter un concept clairement

Les Objets

- Les types d’objet qui existent sont des ***enumération(enum), structure (struct) ou classe (class)***
- Le nom du type d ’objet doit commencer par une lettre Majuscule et des accolades
- Exemple

  - *class Manny {*
    }
  - *struct Moe {*
    }
  - *enum Jack {*
    }

Les Objets

- La portée d'un type d'objet dépend de son emplacement de déclaration :
  - Les types d'objets déclarés au niveau supérieur d'un fichier seront, par défaut, visibles dans tous les fichiers du même module.
  - Il s'agit de l'emplacement habituel pour les déclarations de type d'objet.
  - Parfois, il est utile de déclarer un type à l'intérieur de la déclaration d'un autre type séparé par un espace .
  - C'est ce qu'on appelle un type imbriqué.
  - Un type d'objet déclaré dans le corps d'une fonction n'existera qu'à l'intérieur de la fonction; ces déclarations sont légales mais rares.

Déclaration des Objets 1/4:

- Initialisation
  - Un type d'objet est simplement le type d'un objet. Le but de déclarer le type d'un objet sera généralement (mais pas toujours) pour que vous puissiez créer un objet réel -une instance – de même type. Un initialiseur est une fonction, déclarée et appelée d'une manière spéciale, vous permettant de l'initialisation.
  - Exemple:
    - class Dog {}
    - Créons une instance: Dog() ou bien *let fido = Dog()*

Déclaration des Objets 2/4:

- Propriétés:
  - Une variable déclarée à un niveau supérieur d'une déclaration de type d'objet est une ***propriété***.
  - Par défaut, c'est une propriété d'instance. Une propriété d'instance est étendue à une instance: elle est accessible via une instance particulière de ce type, et sa valeur peut être différente pour chaque type de cet instance
  - Alternativement, une propriété peut être static/class. Pour une énumération ou une structure, il est déclaré avec le mot clé static; pour une classe, on utilise le mot clef class.
  - Une telle propriété appartient au type d'objet lui-même: elle est accessible à travers le type, et il n'a qu'une seule valeur, associée au type.

Déclaration des Objets 2/4:

- Propriétés:
  - Exemple:

```
class Dog {
    let name : String
    let license : Int
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
}
```

```
let fido = Dog(name:"Fido", license:1234)
let spot = Dog(name:"Spot", license:1357)
let aName = fido.name // "Fido"
let anotherName = spot.name // "Spot"
```

```
struct Greeting {
    static let friendly = "hello there"
    static let hostile = "go away"
}
```

Déclaration des Objets 3/4:

- Methods:
  - Une fonction déclarée au niveau supérieur d'un type d'objet est une méthode. Par défaut, c'est une méthode d'instance: elle est appelée en envoyant un message à une instance particulière de ce type. A l'intérieur d'une méthode « Self » est une instance
  - Alternativement, une méthode peut être static/class. Pour une énumération ou une structure, il est déclaré avec le mot clé static; pour une classe, il peut être déclaré par le mot clef class.

Déclaration des Objets 3/4:

- Methods:
  - Exemple

```
class Dog {
    let name : String
    let license : Int
    let whatDogsSay = "woof"
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
    func bark() {
        print(self.whatDogsSay)
    }
    func speak() {
        self.bark()
        print("I'm \(self.name)")
    }
}
```

```
let fido = Dog(name:"Fido", license:1234)
fido.speak() // woof I'm Fido
```

```
    func speak() {
        bark()
        print("I'm \(name)")
    }
```

Déclaration des Objets 4/4:

- Objets Imbriqués:
  - Un type d'objet imbriqué peut être déclaré dans un type d'objet, de manière un imbriqué.
  - Un type d'objet imbriqué n'est pas différent des autres types d'objets, mais les règles de référence semblent être changés de l'extérieur; le type d'objet environnant agit comme un nom-espace, et doit être référencé explicitement pour accéder au type d'objet imbriqué:

```
class Dog {
    struct Noise {
        static var noise = "woof"
    }
    func bark() {
        print(Dog.Noise.noise)
    }
}
```

```
Dog.Noise.noise = "arf"
```

Les Classes

- Une classe est similaire à une structure avec différents éléments
  - ***Type de référence****: Une classes a des types de référence. Cela signifie, entre autres, qu'une instance de classe a deux caractéristiques remarquables qui n'ont pas la même signification pour les instances de struct ou d'énumération:*
    - ***Mutuabilité****: On peut changer la valeur d'une propriété d'instance d'une classe à partir d'un référencement. Même si on définit l'instance de le classe comme une constante (**let**)*
    - ***Multi-référencement:*** *Si on instancie une à plusieurs variables ou la passée en argument, on a une multi référencement avec le même objet*
  - ***Héritange:*** *Une classe peut avoir une super classe. Une classe qui a une sous classe de la super classe, et l'hérite, est un membre de la super classe*

Les Classes:

- class suivi du nom de la class pour créer une classe.
- Les propriétés d'une classe sont déclarés comme les variables et les constantes mais dans le contexte d'une class
- L'héritage se fait après le nom de la classe séparé par « : »

```
class className: SuperClass{
    //Attributs et methodes
}
```

Les Classes

- Exemple simple:

```
class Personne {  var
      nom:String  var
      age:Int

      init(nom:String,age: Int) {
            self.nom= nom
            self.age= age
      }

      func description () -> String {
            return "La personne s'appelle\(nom) et a \(age) ans"
      }
}

var dupont = Personne(nom:"Dupont",age: 18)
dupont.age =17
print(dupont.description())//"La personne s'appelle Dupont et a 17 ans\n"
```

Les Classes

- Exemple d'un héritage:

```
class Femme: Personne{
        var sexe: String

        override init(nom:String, age: Int) {
                self.sexe= "F"
                super.init(nom: nom, age: age)
        }
}

var emilie = Femme(nom: "Emilie", age: 22)  print(emilie.description())//"La personne
s'appelle Emilie et a 22ans\n"
```

Les Classes

- Getters et Setters

```
class TriangleEquilateral          {
      var cote :  Double

      init(cote:  Double)
      {
            self.cote = cote
      }

      var perimetre: Double {  get
      {
        return cote* 3.0
      }
      set{
        cote = newValue/ 3.0
      }
   }
}

var trEq =  TriangleEquilateral(cote:            3.0)

trEq.perimetre       = 27.0
print(trEq.cote)
```

Les Classes

- Instance d’une classe
- Exemple:

```
class Animal{

	func crier() {
		print("Cri de l'animal")
	}
}

var animal = Animal()  animal.crier() //
"Cri de l'animal"
```

Les Classes

- Exemple avec héritage:

```
class Mouton : Animal {

        override func crier() {
                print("Béhh Béhh Béhh")
        }
}

var belier = Mouton()  belier.crier()
//Béhh Béhh Béhh
```

Les Classes

- Comme les interfaces en Java ou PHP
- Héritage multiple

```
protocol MonProtocol {
      func respecteMonProtocol()
}
```

```
class MaClasse: MonProtocol{

      func respecteMonProtocol() {  print("Je respecte le
            protocol")
      }
}
```

Les Classes

- Attention à l'héritage :

```
class MaClasse: MaSuperClasse, MonProtocol{

      func respecteMonProtocol() {  print("Je
            respecte le protocol")
      }
}
```

POO: Enumération

- Une énumération est un type d'objet dont les instances représentent des éléments distincts des valeurs prédéfinies d’une alternative.
- Considérez-le comme une liste de possibilités connues. Une énumération est la façon rapide d'exprimer un ensemble de constantes alternatives les unes aux autres.
- Une déclaration d'énumération comprend est une déclaration de cas. Chaque cas est le nom d'une des alternatives.
- Une instance d’un *enum* représentera exactement une alternative ou un cas

POO: Enumération

- Enum => définir un type commun pour une liste de valeurs personnalisées
- Exemple:

```
enum Genre {
	case HOMME
	case FEMME
}
```

```
var monGenre = Genre.HOMME

switch monGenre {
case .HOMME:
	print("Je suis un homme")
case .FEMME:
	print("Je suis une femme")
}
```

POO: Structure

- Une structure est le type d'objet Swift par excellence. Une énumération, avec son ensemble fixe de cas, est un type d'objet réduit et spécialisé.
- Une classe, à l'autre extrême, se révélera souvent être exagéré; il a certaines fonctionnalités qui manquent à une structure
- Mais si vous n'avez pas besoin de ces fonctionnalités, une structure peut être préférable.

POO: Structure

- Parmi les cent types d'objets déclarés dans l'en-tête Swift, un seul est fondamentalement une classe.
- Une dizaine sont des énumérations. Tout le reste, C’ est à dire la grande majorité d'objet intégré.
- Les types fournis par Swift lui-même sont des structures. Un **String** est une structure. Un **Int** est une structure. Un **Range** est une structure. Un **tableau** est une structure. Etc…
- Cela montre la puissance d'une structure.

POO: Structure

- Fonctionne exactement comme une classe
- Différence:
  - Ne bénéficie pas de l'héritage

```
struct NomDeLaStructure{

}
```

```
// Structures have an auto-generated
(implicit) designated initializer
let namesTable = NamesTable(names: ["Me",
"Them"])
let name = namesTable[1]
print("Name is \(name)") // Name is Them
```

```
// Structures and classes have very similar
capabilities
struct NamesTable {
    let names: [String]

    // Custom subscript
    subscript(index: Int) -> String {
        return names[index]
    }
}
```

Fin de la séquence 3