

DEVELOPPEMENT MOBILE iOS avec Swift

POO avanc é

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



Séquence 4: Swift

- Plan
    - Programmation orientée objet (POO) avancé
        - Polymorphisme
        - Casting
        - Protocoles
        - Génériques
        - Extensions
        - Dictionnaires

Polymorphisme

- Swift obéit aux principes de polymorphisme. C’est le polymorphisme qui transforme un langage basé sur les objets dans un langage orienté objet à part entière.
  - **Substitution :** Partout où un certain type est attendu, un sous-type de ce type peut être utilisé à la place.
  - **Identité Interne:** Le type d'un objet est une question de sa nature interne, quelle que soit la façon dont l'objet est mentionné.

```
class Dog {
}
class NoisyDog : Dog {
}
let d : Dog = NoisyDog()
```

Casting

- Le casting ou Transtypage est un moyen de vérifier le type d'une instance, ou de traiter cette instance comme une super-classe ou sous-classe, mais différente quelque part dans sa propre hiérarchie de classe

```
class MediaItem {
    var name: String
    init(name: String) {
        self.name = name
    }
}

class Song: MediaItem {
    var artist: String
    init(name: String, artist: String) {
        self.artist = artist
        super.init(name: name)
    }
}
```

```
class Movie: MediaItem {
    var director: String
    init(name: String, director: String)
{
        self.director = director
        super.init(name: name)
    }
}

let library = [
    Movie(name: "Casablanca", director:
"Michael Curtiz"),
    Song(name: "Blue Suede Shoes", artist:
"Elvis Presley"),
    Movie(name: "Citizen Kane", director:
"Orson Welles"),
    Song(name: "The One And Only", artist:
"Chesney Hawkes"),
    Song(name: "Never Gonna Give You Up",
artist: "Rick Astley")
]
```

Casting

- Vérification de Type: operator **is:**

```
var movieCount = 0
var songCount = 0

for item in library {
    if item is Movie {
        ++movieCount
    } else if item is Song {
        ++songCount
    }
}

print("Media library contains \(movieCount) movies and
\(songCount) songs")
// prints "Media library contains 2 movies and 3 songs"
```

DownCasting

- Une constante ou une variable d'un type de classe peut faire référence à une instance d'une sous-classe. Si vous pensez que c'est le cas, vous pouvez faire un ***downCast*** de la sous-classe avec un opérateur de conversion (as? Ou as!)

```
for item in library {
    if let movie = item as? Movie {
        print("Movie: '\(movie.name)', dir. \
(movie.director)")
    } else if let song = item as? Song {
        print("Song: '\(song.name)', by \
(song.artist)")
    }
}

// Movie: 'Casablanca', dir. Michael Curtiz
// Song: 'Blue Suede Shoes', by Elvis Presley
// Movie: 'Citizen Kane', dir. Orson Welles
// Song: 'The One And Only', by Chesney Hawkes
// Song: 'Never Gonna Give You Up', by Rick Astley
```

Protocole

- Un protocole est un moyen d'exprimer des points communs entre des types non liés par ailleurs .
- Par exemple, un objet Abeille et un objet Oiseau peuvent avoir besoin de certaines fonctionnalités c’est-à-dire ***voler***. Ainsi, il pourrait être utile pour définir un type de ***Voler***.
- Pour déclarer un protocole, on utilise le mot clef ***@objc protocol***

Protocole

```
@objc protocol TransformShape {
    optional func reshape()
    optional func canReshape() -> Bool
}

class MyShape: Rect {
    var delegate: TransformShape?

    func grow() {
        sideLength += 2

        // Place a question mark after an optional property, method, or
        // subscript to gracefully ignore a nil value and return nil
        // instead of throwing a runtime error ("optional chaining").
        if let reshape = self.delegate?.canReshape?() where reshape {
            // test for delegate then for method
            self.delegate?.reshape?()
        }
    }
}
```

Protocole de Conformité

- La conformité protocolaire dans une classe définit le niveau
- Le nom du protocole est ajouté à côté du nom de la superclasse, séparé par une virgule. S'il n'y a pas de superclasse, il vous suffit d'écrire deux points, suivi du nom du protocole.

```
• Both MyClass and AnotherClass conform to the SampleProtocol
class MyClass: SampleProtocol
{
    // Conforming to SampleProtocol
    func someMethod() {
    }
}

class AnotherClass: SomeSuperClass, SampleProtocol
{
    // A subclass conforming to SampleProtocol
    func someMethod() {
    }
}
```

Générique

- 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

- 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.

Générique

```
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
    }
}
```

```
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
```

Générique

```
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}
}
```

Générique

```
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
```

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
    }

}
```

Extensions

- Ajoute une extra fonctionnalité sur un type existant
- Exemples: les variables et les fonctions
- Les extensions peuvent contenir :
  - des fonctions
  - et des variables get / constantes. Ils sont dans le format

```
extension ExtensionOf {
    //new functions and get-variables
}
```

```
extension String {
    func length() -> Int {
        return self.characters.count
    }
}

"Hello, World!".length() // 13
```

```
extension String {
    var length: Int {
        get {
            return self.characters.count
        }
    }
}

"Hello, World!".length // 13
```

Extensions

- Extension en construisant un nouveau type

```
extension Int {
    var customProperty: String {
        return "This is \(self)"
    }

    func multiplyBy(num: Int) -> Int {
        return num * self
    }
}

print(7.customProperty) // "This is 7"
print(14.multiplyBy(3)) // 42
```

Extensions

- Exemple

```
extension Square: CustomStringConvertible {
    var description: String {
        return "Area: \(self.getArea()) - ID: \
(self.identifier)"
    }
}

print("Square: \(mySquare)")
```

Extensions

- Extension en construisant un nouveau type

```
extension Int {
    var customProperty: String {
        return "This is \(self)"
    }

    func multiplyBy(num: Int) -> Int {
        return num * self
    }
}

print(7.customProperty) // "This is 7"
print(14.multiplyBy(3)) // 42
```

Dictionnaires

- **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>()
```

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

Dictionnaires

- 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"
```

Dictionnaires

- 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"
```

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

CE QU'IL FAUT RETENIR

- Swift 3 :
  - Variable
    - Une variable mutable se déclare avec le mot clé ***var***
    - Une variable immutable se déclare avec le mot clé ***let*** (constante)
    - Une optionnelle est variable qui peut contenir ***nil*** comme valeur
  - Fonction
    - Une fonction se définit avec le mot clé ***func.***
    - Une fonction peut prendre zéro à plusieurs arguments, qui peuvent être de n'importe quel type, ou bien même des ***closures***.
  - Classe
    - Une classe se crée avec le mot clé ***class***.
    - Elle peut aussi hériter d'une seule classe parente et implémenter un nombre illimité de protocols.
  - Conditionnement & boucle
    - ***if*** & ***switch*** sont utilisés pour les conditions
    - ***for***, ***while*** & ***repeat-while*** sont utilisés pour les boucles

Fin de la séquence 4