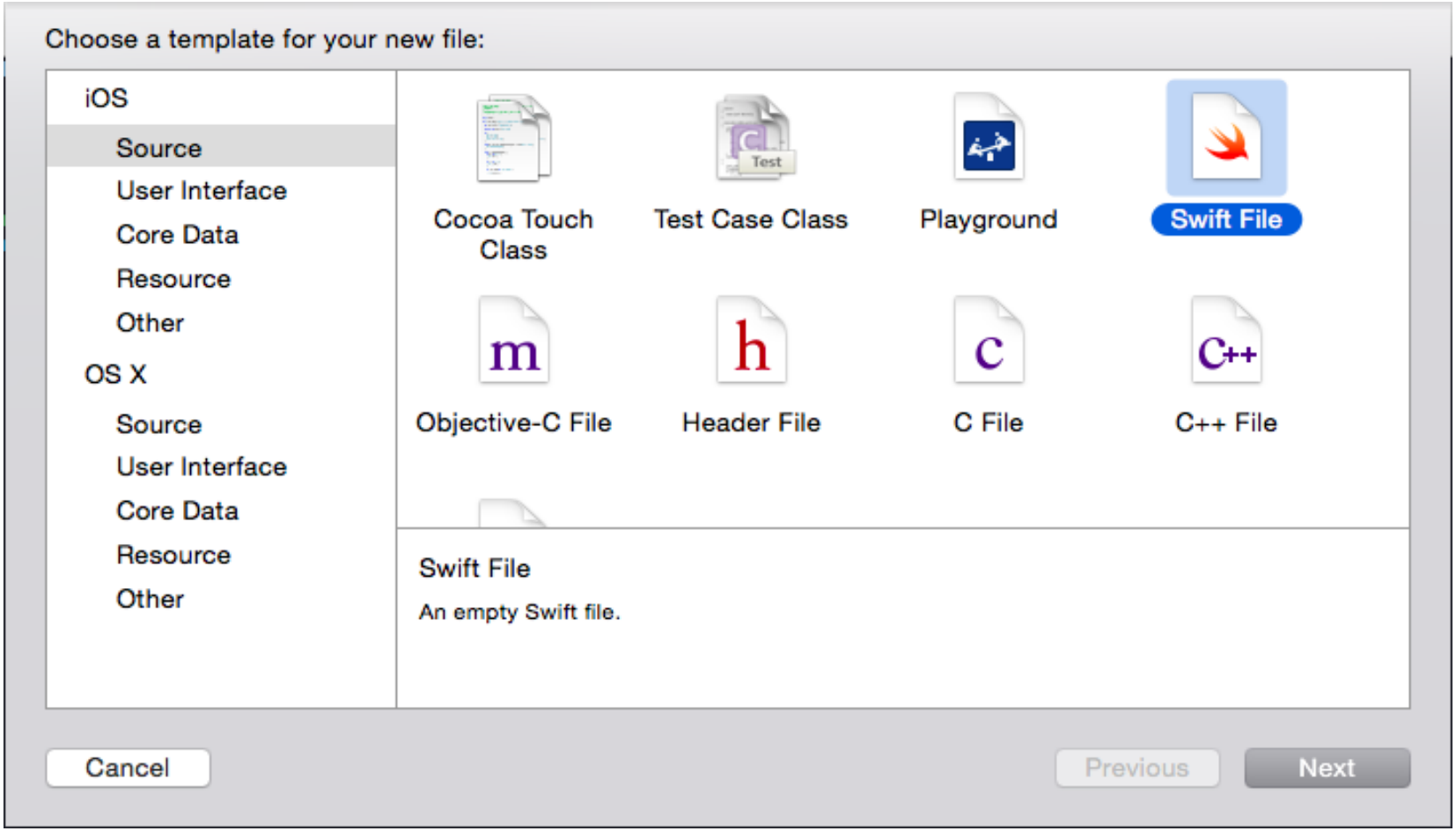
Partie II

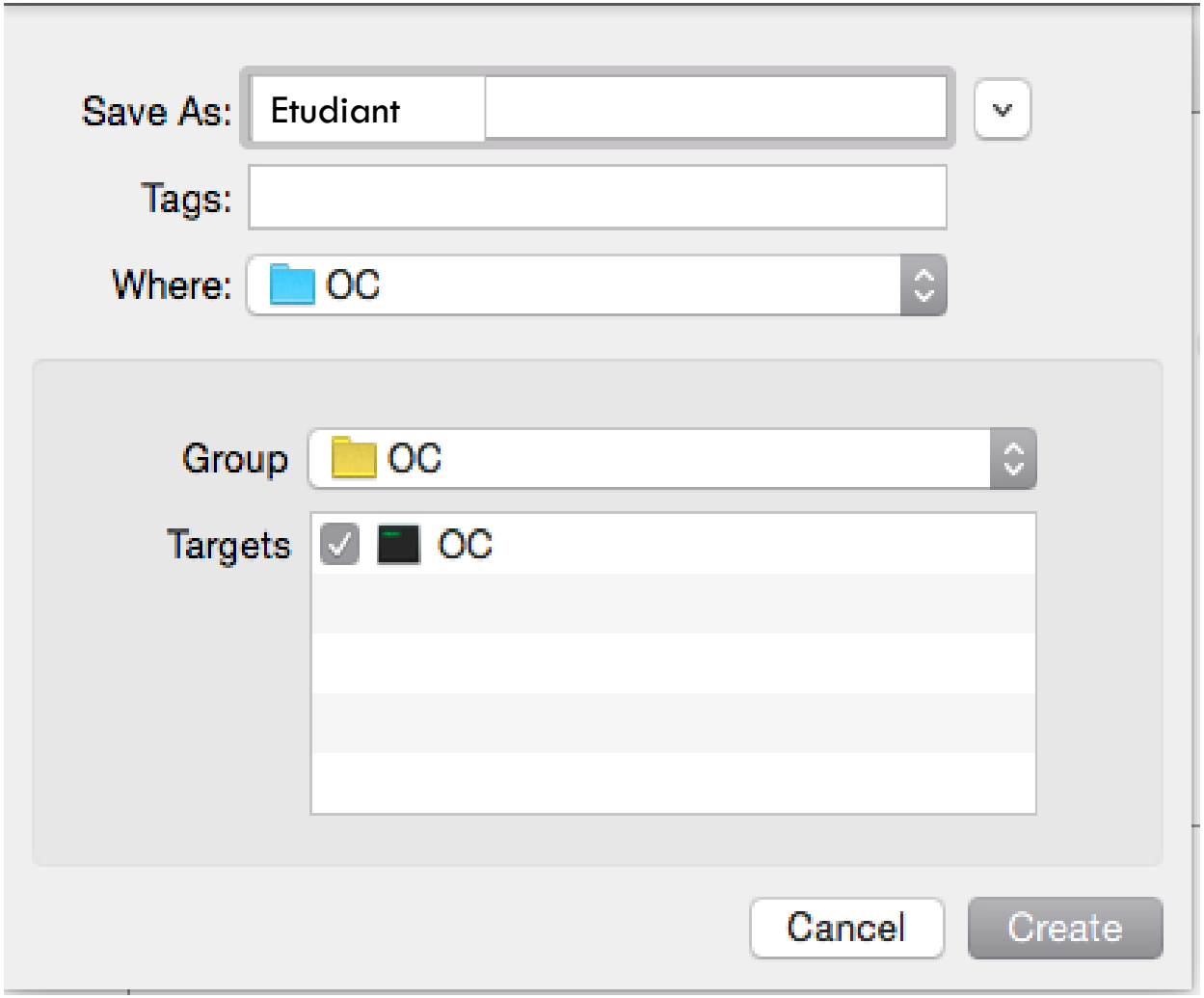
LA PROGRAMMATION ORIENTÉE OBJET EN SWIFT

LES CLASSES ET LES STRUCTURES



Création d'un nouveau fichier Swift

LES CLASSES ET LES STRUCTURES



LES CLASSES ET LES STRUCTURES

```
class Etudiant {
    var nom: String
    var prenom: String
    var age: Int
    var adresse: String
}
```

erreur !! il faut définir des valeurs par défaut à une classe, c'est obligatoire

```
class Etudiant {
    var nom: String      = "Alami"
    var prenom: String = "Mohammed"
    var age: Int             = 21
    var adresse: String  = "5 rue liberte Rabat"
}
```

LES CLASSES ET LES STRUCTURES

```
class Etudiant {
    var nom: String      =  "Alami"
    var prenom: String = "Mohammed"
    var age: Int              = 21
    var adresse: String  =  "5 rue liberte Rabat«


    func sinscrire() {
     }

     func passerExamen() {
     }
    func demenager() {
    }
}
```

LES CLASSES ET LES STRUCTURES

```
class Etudiant {
    var nom: String      =  "Alami"
    var prenom: String = "Mohammed"
    var age: Int              = 21
    var adresse: String  =  "5 rue liberte Rabat "
    var statut : String  = " N"
    func sinscrire() {
      if self.statut == "N"  {   statut = "O" }
     }

     func passerExamen() {
     }
    func demenager(nouvelleAdresse: String) {
     self.adresse = nouvelleAdresse
    }
}
```

LES CLASSES ET LES STRUCTURES

```
let etudiant = Etudiant()
let etudiant = Etudiant()
print("Nom de l'étudiant  au début : " + etudiant.nom)
etudiant.nom = "Farah"
print("Le nouveau nom est maintenant : " + etudiant.nom)

// On peut réaliser des actions propres à l' étudiant

print("Mon âge est \(etudiant.age) ")
print("Actuellement j'habite à cette adresse : " + etudiant.adresse)
etudiant.demenager("12 rue Casablanca")
print("J'habite dorénavant à cette adresse : " + etudiant.adresse)
```

LES CLASSES ET LES STRUCTURES

- *La surcharge de méthode*

La surcharge de méthode consiste à déclarer une nouvelle méthode portant le même nom mais avec :

- ✓ le même nombre de paramètres, mais au moins un type de ces paramètres est différent que celle de la méthode existante ;
- ✓ un nombre de paramètres différent ;
- ✓ les deux.

```
    func affiche() {

        print("Nom : " + self.nom)
        print("Prénom : " + self.prenom)
        print("Age : \(self.age)")
        print("Adresse : " + self.adresse)
    }

    func affiche(valeurAAfficher: String) {

        switch valeurAAfficher {
            case "Nom":
                print("Nom : " + self.nom)

            case "Prénom":
                print("Prénom : " + self.prenom)
            case "Age":
                print("Age : \(self.age)")
            case "Adresse":
                print("Adresse : " + self.adresse)

            default:
                print("Nous n'avons pas pu afficher ce que vous avez demandé.")
        }
    }
}

Etudiant.affiche()
Etudiant.affiche(" age ")
```

Le constructeur

```
class Etudiant {
    // On peut maintenant supprimer les valeurs par défaut
    var nom: String
    var prenom: String
    var sexe: String
    var age: Int
    var adresse: String

    // Constructeur
    init() {
            }
```

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

```
let  etudiant  = Etudiant(nom :"Salhi", prenom = "Ahmed", age : 20 , adresse = "94 rue Saada")
```

Le concept des propriétés

- les propriétés stockées sont associées à une valeur
- les propriétés calculées sont associées à un calcul

une propriété admet deux actions :

1. une action pour récupérer la valeur contenue, on va appeler cette action **get** *(récupérer en anglais)*
2. une action pour modifier la valeur contenue, on va appeler cette action **set** *(modifier en anglais)*

:

```
class Carré {
var longueur = 2
var périmètre: Int {
        get {
            return  longueur*3
        }
        set {
          }
      }
}
```

Entre les accolades se trouvent deux sortes de sous-fonctions :

- **get** : c'est l'action qui permet de récupérer une valeur, on appelle ça le **getter**. Elle se comporte exactement comme une fonction avec une valeur de retour de type Int.

- **set** : c'est l'action qui permet de modifier notre valeur, on appelle ça le **setter**. Pour l'instant cette fonction est vide. Telle quelle, elle ne change rien à ce qu'il se passe lorsqu'on modifie la valeur de la variable périmètre.

```
var périmètre: Int {
        get {
        return longueur * 4
        }
        set {
        longueur = newValue / 4
        }
}
```

Le setter se comporte exactement comme une fonction qui a pour paramètre `newValue`.
`newValue` contient la nouvelle valeur que l'on est en train de donner au périmètre.

Observation des propriétés

deux méthodes pour écouter des propriétés **stockées** :

- **willSet** : cette méthode est appelée juste avant la modification de la propriété.
- **didSet** : cette méthode est appelée juste après la modification de la propriété.

```
var longueur = 1 {
        willSet {
        print("Le carré va changer de taille")
        }
        didSet {
            if oldValue > longueur {
            print("Le carré a rapetissé")
            } else {
            print("Le carré a grandi")
            }
        }
}
```

set : pour les propriétés calculées

willSet et didSet : pour les propriétés stockées

Les structures

```
struct   NomStructure {
    // Attributs et méthodes
}
```

Les structures

```
etudiant = Etudiant(nom: "Alami", prenom: "Ahmed", age:
   27, adresse: "94 rue rabat")
var  etudiant1 = etudiant
etudiant.age = 12
etudiant.affiche("Age")
etudiant1.affiche("Age")
```

```
Age : 12
Age : 12
```

Les structures

```
struct Personne {

    var nom: String
    var age : String

    func quiEstCe() {
        print("Nom : " + self.nom + ", age : " + self.age)
    }
}

var  personne = Personne(nom: "Alami", age: 20)
var personne2 = personne

personne2.nom = "Lotfi"

personne.quiEstCe()
```

```
Nom : Alami Age  :  20
Nom  : Lotfi  Age  :  20
```

L'HÉRITAGE

```
class Etudiant : Personne {

}
```

:

```
let personne = Personne(nom: "Alami", prenom: "Ahmed", age: 20, adresse: "94 casa")

let  etudiant = etudiant(nom: "Alami", prenom: "Ahmed", age: 20, adresse: "94 rue casa")

personne.affiche()
Etudiant.affiche()
```

```
class Etudiant : Personne {

    var  modules: [String] = [ ]        // Par défaut, le tableau est vide
    func passerControle(nomModule: String) {
    }
    func participerStage (nomSociete: String) {
    }
}
```

L'HÉRITAGE

```
class Etudiant : Personne {

    var  modules: [Module] = [ ]        // Par défaut, le tableau est vide
    func passerControle(nomModule: Module) {
    }
    func participerStage (nomSociete: Societe) {
    }
}
```

Exercice

Dans la fonction Main :

- ✓*Créer des étudiants*
- ✓*Affecter des modules à chaque étudiant*
- ✓*Lister les différents modules de chaque étudiant*

LE POLYMORPHISME

```
class Etudiant : Personne {

	func afficher() {
		super.afficher()
		Print(" les modules  sont :\(modules) " )
		}
	}
	func participerStage (nomSociete: Societe) {
	}
}
```

```
class Etudiant : Personne {
var  modules:[String]
init(nom: String, prenom: String, sexe: String, age: Int, adresse: String, modules: [String])
	 {
	Self.modules = modules
	super.init(nom: nom, prenom: prenom, sexe: sexe, age: age, adresse: adresse)
	}
}
```

Récapitulatif

- On peut réaliser un héritage si l'on peut dire que « B est un A », B et A étant deux classes différentes.
- Une classe B qui hérite d'une autre classe A possédera tous les attributs et méthodes de A.
- Une classe héritée peut avoir des attributs et méthodes supplémentaires que ceux de sa classe mère.
- On peut faire hériter une classe qui est déjà héritée (et ainsi de suite) mais on ne peut faire hériter deux classes à une même classe.
- Il est possible grâce au polymorphisme de redéfinir ou modifier le comportement d'une méthode.

Les Protocoles

Syntaxe de déclaration :

```
:
protocol NomDeMonProtocole {
}
```

```
:
protocol ProtocoleVéhicule {

    func  démarrer()
    func  accélerer()
}
```

Les Protocoles & Héritage

Utilisation d'un protocole :

```
Class  Voiture :Vehicule, ProtocoleVéhicule {

        instructions….
    }
}
```

```
Protocole  ProtocoleVéhicule1 : ProtocoleV2 {
Class  voiture  : ProtocoleVéhicule1 {
  //…..

}
```

LES ÉNUMÉRATIONS

Déclaration d'une énumération:

```
enum NomEnumeration {
// Liste des différents cas ici }
```

:

```
enum Langage {
case Swift
case Java
case C++
case C
}

Ou
enum Langage {case Swift , Java , Php, C  }
```

LES ÉNUMÉRATIONS

Utilisation d'une énumération:

```
var lanaggeuilise = Langage.Swift
    langageutilise = .java
```

```
var langageUtilise = Langage.Swift
switch langageUtilise {
case .Swift:
print("le langage Swift.")
case .Java:
print("  le Java.")
case .C++:
print(" le C++.")
case .C:
print(" le langage C.")
}
```

LES ÉNUMÉRATIONS

Valeurs associées à une énumération :

```
enum Langage {
case Swift ( String , String)
case Java (String)
case C++ (String)
case C (String)
}
```

```
 var  lanaggeuilise  =  Langage.Swift("Le langage Swift",  "3 modialement")
      langageutilise  =  .java(("Le langage java")
```

LES ÉNUMÉRATIONS

Valeurs associées à une énumération :

```
var lanaggeuilise = Langage.Swift("Le langage Swift" ,"N°3")
    langageutilise = .java(("Le langage java")
```

```
switch langageUtilise {
case let .Swift(description, niveau):
print(description +"  "+ niveau)
case var .Java(description) :
print( description)
default :
print("  autre langage.")

}
```

LES EXTENSIONS

Ajout d'une méthode:

```
.
extension Int {
func carre() -> Int {
return self * self
}
func cube() -> Int {
return self * self * self
}
func afficheCarreEtCube() {
print("\(self.carre()), \(self.cube())")
}
}
5.afficheCarreEtCube()
print(4.cube())
```

IOS