

# Formation du langage Swift



## Les bases du langage Swift

M<sup>me</sup> Kathia CHENANE

# Plan

- Présentation de Swift
  - Swift
  - Historique
  - Comparaison avec l'Objective C
- Installation des outils de développement
  - Installation Xcode
  - Utiliser Xcode
  - Programmer en Swift avec Xcode
- Les variables et opérations
  - Exercice Pratique 1

# Plan

- Les conditions
  - Exercice Pratique 2
- Les boucles
  - Exercice Pratique 3
- Les tableaux et les dictionnaires
  - Exercice Pratique 4
- Les fonctions et closures
  - Exercice Pratique 5
- TP

# Plan

- Les conditions
  - Exercice Pratique 2
- Les boucles
  - Exercice Pratique 3
- Les tableaux et les dictionnaires
  - Exercice Pratique 4
- Les fonctions et closures
  - Exercice Pratique 5
- POO
- TP

# Présentation du langage Swift <sup>1/3</sup>

## Swift

- Langage de programmation créé par Apple en 2014
- Créé pour élaborer des applications sous le système d'application iOS
- Mise à jour de Swift régulières, la dernière en date juillet 2020 5.2
- Langage plus rapide comme son nom l'indique

# Présentation du langage Swift <sup>2/3</sup>

## Historique

- Développement de Swift a commencé avec Chris Lattner un superviseur du département de « developer Tools » chez Apple
- Lancement le 2 juin 2014

# Présentation du langage Swift <sup>3/3</sup>

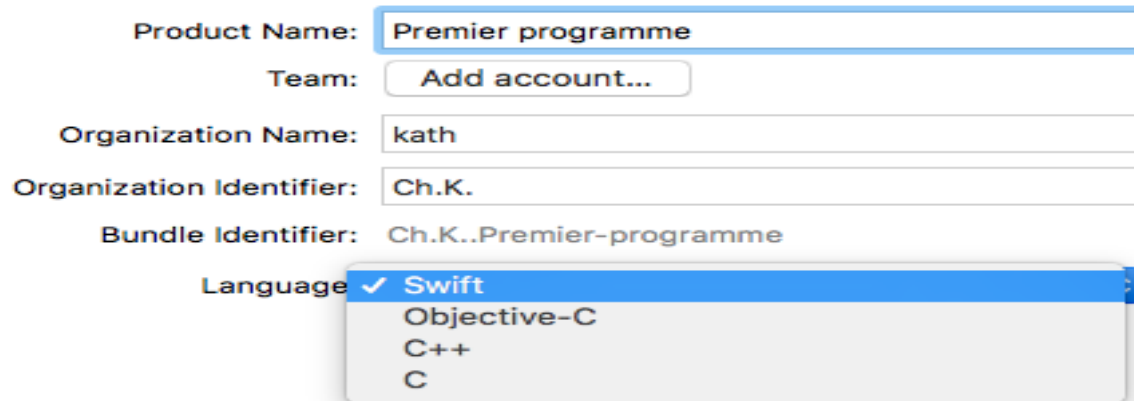
## Comparaison avec l'Objective C

- Inspiré de l'Objective C, C#, Python
- Plus intuitif et rapide
- Parmi les changements de syntaxe :
  - Condition de if sans parenthèses avec une accolade à droite.
  - Disparition du break pour Switch
  - Le mot func pour définir une fonction.
  - Point-virgule optionnel.

# Outils de développement

## Premier programme avec Xcode

Nommer le projet et choisir le langage de programmation Swift :



The image shows a screenshot of the Xcode project creation dialog. The 'Product Name' field is highlighted with a blue border and contains the text 'Premier programme'. The 'Team' field has a button that says 'Add account...'. The 'Organization Name' field contains 'kath'. The 'Organization Identifier' field contains 'Ch.K.'. The 'Bundle Identifier' field contains 'Ch.K..Premier-programme'. The 'Language' dropdown menu is open, showing a list of programming languages: Swift (selected with a checkmark), Objective-C, C++, and C.

Product Name: Premier programme

Team: Add account...

Organization Name: kath

Organization Identifier: Ch.K.

Bundle Identifier: Ch.K..Premier-programme

Language: ☒ Swift  
☐ Objective-C  
☐ C++  
☐ C



# **Les bases du langage Swift**

# Les variables et opérations

## Déclaration de variable :

En swift la déclaration d'une variable est précédée du mot clé *var* .

- **Déclaration implicite** : Sans préciser le type de données

***var** variable = 2*

- **Déclaration explicite** : En précisant le type de données dans notre exemple Int

***var** variable : Int = 2*

# Les variables et opérations

## Les différents types :

**Int** : integer, entier

**Float**: float, nom à virgule sur 4 octets

**Double**: double, nombre à virgule sur 8 octets

**Bool**: boolean, valeurs (*true, false* )

**String**: Chaîne de caractères

**Character**: caractère (char en C )

# Les variables et opérations

## Déclaration de constantes :

La déclaration des constantes est quasi identique aux variables **sauf que** le mot clé utilisé est *let* à la place de *var*

Contrairement aux variables qui peuvent être modifiées (muables) les constantes sont immuables.

- Déclaration implicite : Sans préciser le type de données  
*let constante = 2*
- Déclaration explicite : En précisant le type de données  
dans notre exemple Int  
*let constante: Int = 2*

# Afficher le contenu

- Pour afficher le contenu d'une variable ou une constante en utilisant la fonction prédéfinie

***Print()***

Exemple :

```
let a = 2
```

```
print("le contenu de",a,"donc a est un entier")
```

```
let chaine = "bonjour"
```

```
print("le contenu de ma variable est "+chaine+"!!")
```

# Afficher le contenu 1/2

- **Interpolation** grâce à *print()*

Exemple :

```
let a = 2
```

```
print("le contenu de \a) donc a est un entier")
```

```
let chaine = "bonjour"
```

```
print("le contenu de ma variable est \chaine!!!")
```

# Les chaînes de caractères

Les chaînes de caractères peuvent inclure les caractères spéciaux suivants :

- `\0` nul
- `\\` antislash
- `\t` tabulation horizontale
- `\n` saut de ligne
- `\r` retour en début de ligne
- `\` » guillemet
- `\'` apostrophe

# Importe / commentaires <sup>1/2</sup>

- Commentaires :

// : Commentaire sur une seule ligne

/\* \*/ : Commentaire sur plusieurs lignes

- Fichier import Foundation

import Foundation

– Au début du fichier **main.Swift**

– Permet d'accéder à NSObject et ses sous-classes



# Importe / commentaires <sup>2/2</sup>

- **Fichier import Foundation (suite)**
- Accéder aux classes essentielles qui définissent le comportement des objets de base, les types de données et services du système d'exploitation.
- Incorporer les modèles de conception et des mécanismes qui rendent vos applications plus efficaces et robustes.

# Exercice pratique 1

- Lancez et créez un projet nommé ExercicesCours
- Ecrire un petit programme dans main.swift qui déclare, initialise et affiche le contenu des différents types vus précédemment.

# Les tuples <sup>1/2</sup>

- Les tuples sont un type de variables contenant plusieurs valeurs.
- **La syntaxe**  
*let* tuple (valeur1, valeur2, valeur3, ...)  
Ou *var* tuple (valeur1, valeur2, valeur3, ...)

# Les tuples 2/2

## Exemple

```
//déclarer un tuple perosonne de deux valeurs
var personne = (prenom:"", nom: "")
//nitilaiser les tuples
personne.nom="toto"
personne.prenom="Dupond"
//afficher le contenu d'un tuple
print("le nom et prénom de cette personne :  \(personne.nom)  \(personne.prenom) ")
// affecter les valeurs d'un tuple à un autre tuple
var (valeur1,valeur2) = personne
//afficher le contenu
print("le nom et prénom de cette personne :  \(valeur1) et \(valeur2) ")
```

# Réaliser un test : conditions logiques

Opérateurs de comparaison :

Syntaxe	emploi	vrai si
==	val1 == val2	val1 est égal à val2
!=	val1 != val2	val1 est différent de val2
<	val1 < val2	val1 strictement inférieur à val2
<=	val1 <= val2	val1 inférieur ou égal à val2
>	val1 > val2	val1 strictement supérieur à val2
>=	val1 >= val2	val1 supérieur ou égal à val2

Attention à l'opérateur d'égalité ==, il ne faut pas le confondre avec l'opérateur d'affectation =

# Opérateurs logiques

En fait, une condition est une expression qui donne une valeur de type vrai ou faux.

Il existe des opérateurs logiques permettant de construire des expressions logiques (de même que l'on construit des expressions mathématiques avec des opérateurs mathématiques).

Arithmétique des valeurs logiques :

opérateurs ET, OU et NON

ET et OU : composent 2 valeurs logiques

NON : s'applique à une valeur logique

# Opérateurs logiques

Tables de vérité des opérateurs logiques. Soient A et B deux conditions, qui peuvent prendre les valeurs VRAI ou FAUX.

- L'expression A ET B a pour valeur VRAI ssi A et B ont pour valeur VRAI en même temps, sinon A ET B vaut FAUX.
- L'expression A OU B a pour valeur VRAI si A est VRAI ou si B est VRAI.
- L'expression NON A a pour valeur VRAI si A a pour valeur FAUX, et vaut FAUX si A a pour valeur VRAI.

Traduction en C :

ET :    &&

OU:    ||    (2 barres verticales, Shift+alt+L)

NON    :    !

# Opérateurs logiques

Remarque sur les parenthèses des expressions logiques :

&& est prioritaire sur ||.

*Avec Swift les parenthèses ne sont pas obligatoires mais si vous définissez une priorité, il faut systématiquement utiliser des parenthèses (les plus prioritaires) pour construire les expressions.*



# Syntaxe de l'instruction **if**

Réaliser un test simple : si une condition est vraie alors faire une instruction.

```
if condition
```

```
{
```

```
Instruction
```

```
} // Les accolades sont obligatoires même pour une  
seule instruction
```

ou

si la condition est fausse, l'instruction (ou le bloc) ne sera pas effectué.

# l'instruction **if** dans un programme

Programme exemple : test d'une valeur

```
if u > b && u < c
{
    print("u appartient à l'intervalle ouvert u,c ")
}
```

## **Remarques:**

- Les accolades sont obligatoires
- Les parenthèses ne sont pas obligatoires
- Bien espacer les opérateurs de comparaison des variables

# L'alternative : **if ... else**

Syntaxe :

```
if condition
{
    instructions; /* si la condition est vraie */
}
else
{
    autres instructions; /* si elle est fausse */
}
```

# L'alternative dans un programme

```
if mystere != solution
{
    print("Révissez vos classiques !\n ")
}
else    /* si la condition est fausse */
{
    print("Bravo !\n ")
}
```

# Condition ternaire

```
var res : Bool
```

```
if mystere != solution  
  {  
    res= true  
  }  
else  
  {  
    res= false  
  }
```

```
// la version ternaire
```

```
res= mystere != solution ? true: false
```

# Conditions et alternatives

## **Avec une condition de type**

`val1 == val2`

`val1 != val2`

`val1 < val2`

`val1 <= val2`

de même pour `>` et `>=`

## **Les instructions du bloc lié au else seront faites si**

`val1 différent de val2`

`val1 est égal à val2`

`val1 supérieur ou égal à val2`

`val1 strictement supérieur à val2`

# Instruction de sélection (Switch)

Dans le cas où nous avons énormément de conditions imbriquées, il est possible de simplifier en utilisant un switch :

**La syntaxe :**

```
switch choix
{
    case 1 : //Le choix saisi est 1
    case 2,3,4 ://Le choix est soit 2 ou 3 ou 4
    case 5...10 ://Le choix va de 5 à 10 inclus
    case 5..<10 ://Le choix va de 5 à 9 inclus
    default : //choix par défaut
}
```

# Exercice appliqué 2

**if...else** permet de traiter deux conditions exclusives l'une de l'autre.  
Mais pour 3 conditions exclusives ? Ou plus ?

Exemple : Résolvez une équation de  $d^2$  dans  $\mathbb{R}$ , en tenant compte du cas  $\Delta=0$  : 3 cas à traiter (*initialisez les variables*)

si  $\Delta < 0$  : pas de solutions

si  $\Delta = 0$  : une racine réelle double

si  $\Delta > 0$  : deux racines réelles distinctes

3 conditions exclusives :

$\Delta$  est forcément strictement négatif ou strictement positif ou nul !

Pour une précision exemple:

```
var a : Float = 22.33333
```

```
print("valeur",String(format: "%.3f ici",a))
```



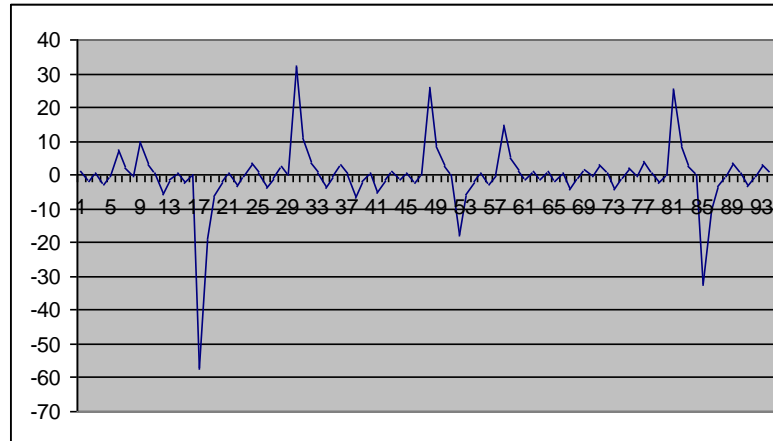
# Les boucles / Itérations

Opération à répéter un certain nombre de fois :

exemple des suites mathématiques :

$$\left. \begin{array}{l} U_0 \text{ donné,} \\ U_{n+1} = f(U_n) \end{array} \right\}$$

exemple :  $U_0 = 1, U_{n+1} = \frac{1}{3} \cdot U_n - \frac{2}{U_n}$



# Les itérations

On veut la valeur de U6 : calcul par suite d'instructions simples

illustration :

```
var terme_U = 1.0    //initialisation du U avec U0
terme_U = (terme_U/3.0)-(2.0/terme_U) // calcul U1
terme_U = (terme_U/3.0)-(2.0/terme_U) // calcul U2
terme_U = (terme_U/3.0)-(2.0/terme_U) // calcul U3
terme_U = (terme_U/3.0)-(2.0/terme_U) // calcul U4
terme_U = (terme_U/3.0)-(2.0/terme_U) // calcul U5
terme_U = (terme_U/3.0)-(2.0/terme_U) // calcul U6

print("U6 = \n",terme_U)
```

# Les itérations

6 fois la même instruction !

Si l'on veut calculer  $U_{150}$ , on devra répéter cette instruction 150 fois !

Réécrire le programme pour chaque valeur différente !

Autre exemple : l'ordinateur choisit un chiffre, l'utilisateur doit le deviner en proposant des valeurs. A chaque proposition, l'ordinateur indique si la proposition faite est supérieure, inférieure ou égale.

# Les itérations

Avec des if .. else on ne fait les tests qu'une fois ! Il faudrait faire ces tests jusqu'à ce qu'on trouve effectivement le bon résultat !

Il existe en Swift des instructions permettant des itérations : répétition d'instructions quand une condition est satisfaite :

**while()** : tant que

**repeat... while()** : faire tant que

syntaxe :

**while** condition {

instruction

}

# Les itérations

Avec **repeat...while**, on fait au moins une fois les instructions du corps de la boucle

Avec **while**, on peut ne pas faire les instructions de la boucle, si la condition est fausse dès le premier test !.

Exemple d'applications : saisie sécurisée d'une valeur.

Pour tester si une valeur est correcte, il fait d'abord la saisir !

Si cette valeur est incorrecte, on la ressaisira : on utilise alors une boucle **repeat...while**, puisqu'il faut faire les instructions (ici la saisie) AVANT de faire le test de la condition !

# Répétition avec la boucle for <sup>1/2</sup>

Dernier type de boucle : la boucle for, qui sert essentiellement à répéter une ou des instructions un nombre de fois déterminé.

Conceptuellement très proche de la boucle while.

En incrémentant de 1 :

Exemple :

```
for _ in 1...5 {  
    print("Hello World")  
}
```

# Répétition avec la boucle for

**En incrémentant avec d'autre pas > 1 :**

**Exemple :** en utilisant un pas de 2 :

```
for i in stride(from: 0, to: 10, by: 2)
{
    print("i= ", i)
}
```

# Exercice appliqué 5

Ecrire un programme qui permet d'afficher les  $n$  nombres premiers par pas de 3 en utilisant les différentes boucles vues précédemment



# **Les collections : tableaux et dictionnaires**

# Tableaux

## Les tableaux (array)

Les tableaux contiennent des valeurs de même type.

## Déclaration d'un tableau :

Déclare un tableau de type chaine de caractères

```
var tableauChaines = [String]()
```

## Initialisation d'un tableau :

```
let tableauEntier = [1,2,3,4,5,6];
```

# Tableaux

## Initialiser les valeur d'un tableau de taille 4 à 2.0

```
var tableauDoubles = [Double](repeating:2.0,count:4)
```

Récupérer la taille d'un tableau : On utilise la fonction *count*

```
var taille = tableauDoubles.count // la taille=4
```

Ajouter un élément en début de tableau : Ajouter au début du tableau

```
tableauDoubles = [9.8,17.9] + tableauDoubles
```

# Tableaux

**Ajouter un élément fin de tableau:** Ajouter au début du tableau

```
tableauDoubles += [3.2,0.9]
```

**Ajouter un seul élément en fin de tableau (concaténation)**

```
tableauDoubles.append(8.0)
```

# Tableaux

**Remplacer un élément existant:** remplacer un élément existant dans le tableau on utilise l'indice, par exemple :

tableauDoubles[2]=7.9

**Remplacer toute une plage d'éléments :**

tableauDoubles [1...2] = [0.0]

# Tableaux

## Supprimer un élément à l'indice i

tableauDoubles.*remove*(at: i)

## Supprimer tous les éléments

tableauDoubles.*removeAll*()

# Tableaux

Vérifier si un tableau est vide : appel de la méthode isEmpty()

```
if tableauDouble.isEmpty{  
    print("vide")  
}else{  
    for elment in tableauDouble  
    {  
        print (elment )  
    }  
}
```

# Tableaux

## Insérer un élément à l'indice i

### Syntaxe :

tableau.insert(newElement:Element , at: Int)

### Exemple

tableauDouble.*insert*(6.9, *at*: 0)



# Les dictionnaires

Dictionnaire est une collection qui nous permet d'enregistrer une valeur en spécifiant une clef unique.

Dictionnaire (cle, valeur)

## Déclarer un dictionnaire

```
var dictionnaire = [Int: String]()
```

## initialiser un dictionnaire

```
dictionnaire = [1:"papier",2:"stylo",3:"pc"]
```

# Les dictionnaires

## Parcourir un dictionnaire

```
for (cle, valeur) in dictionnaire {  
    print("\(cle)  \(valeur)")  
}
```

## Afficher la valeur d'une clef

```
print("\(dictionnaire[1])")
```

## Mettre à jour la valeur d'une clef

```
let maj= dictionnaire.updateValue("des stylos", forKey:  
2)
```

# Les dictionnaires

## Supprimer la valeur d'une clef

let removedValue = dictionnaire.removeValue(forKey: 1)

# **Les fonctions et closures**

# Les fonctions

- Une fonction est un morceau de code qui effectue une tâche spécifique.
- Une fonction a *un nom* qui décrit son but, ce nom est utilisé pour *appeler la fonction* pour effectuer la tâche en cas de besoin.
- Des données peuvent être fournis à une fonction en transmettant des paramètres.
- La fonction peut fournir des données de retour comme résultat.
- Les fonctions/procédures en Swift sont précédées par le mot clef *func*.

# Les fonctions

*Procédure : est une fonction qui ne retourne aucun résultat*

**Exemple** de fonction (en Swift on parle de fonction ) qui effectue un affichage sans paramètre en entrée

```
func affiche() { // affiche 1 à 10
  for i in 1 ... 10
  {
    print ( i )
  }
}
```

# Les fonctions

Une fonction qui prend des paramètres en entrée sans rien retourner

## Syntaxe

```
func nomFonction(parametre1:Type1 , parametre2:Type2, .....)  
{  
    // instructions  
}
```

*Exemple* de fonction qui affiche la somme de deux entiers

```
func somme (a : Int , b : Int) {  
    print("la somme de \a) + \b) = \a+b)")  
}
```

# Les fonctions

Une fonction qui prend des paramètres en entrée et retourne une valeur

## Syntaxe

```
func nomFonction(par1:Type1 , parN:TypeN) -> TypeDeRetour  
{  
    // instructions  
    return valeurDeTypeRetour  
}
```

*Exemple* de fonction qui retourne la somme de deux entiers

```
func somme (a : Int , b : Int) -> Int {  
    return a+b // On retourne la somme de a et b  
}
```



# Les fonctions

**Appel des fonctions définies** : L'appel des fonctions se fait après la déclaration – définition

**Exemple** : Appel des précédentes fonctions définies

affiche()

somme(a: 3, b: 4) // **Remarque** : On doit spécifier les noms des paramètres avant de leur attribuer une valeur

//Appel d'une fonction qui retourne la somme des entiers

```
print("La somme retournée est \"(sommeEntiers(a: 3, b: 7))\"")
```

# Les closures

Les closures (fermeture en français) sont des blocs autonomes de code que, vous pouvez passer à des fonctions comme paramètres ou les stocker comme des variables ou des constantes afin de les utiliser plus tard ou encore les retourner par une autre fonction (On parle alors de high-order function).

Une closure ressemble à une fonction, mais est bien plus puissante parce qu'elle peut être anonyme (sans nom).

# Les closures

Il y a trois types de closures dans Swift, et nous avons déjà vu deux d'entre eux :

**Les fonctions globales** : sont des closures qui ont un nom et ne capturent aucune valeur.

**Les fonctions imbriquées** : sont des closures qui ont un nom et peuvent capturer les valeurs de fonctions qui les englobent.

**Les closures expression (expression de fermeture)**, qui n'ont pas un nom, mais peuvent capturer les valeurs de leurs contextes.

# Les closures

## Exemple d'exécution

```
/* Déclarer une closure qui ne prend pas de paramètre
et renvoie un string dans une variable */
var affiche: () -> (String) = {
    return "Bonjour"
}

// Appel de la closure
print (affiche()) // Bonjour

// Déclarer une closure qui prend en paramètre deux entiers et renvoie un entier
var multiplication: (Int,Int) -> (Int) = { x , y in
    return x * y
}
print(multiplication(2,3)) // 12

// affecter une closure à une variable
var alsoDouble = multiplication
print(alsoDouble(3,2)) // 12
```

# **Programmation Orientée Objet**

# Plan

Classes

Instanciación d'objets

Structures

Héritage

Surcharge de classe

# Les Classes

## Définition:

Une classe est semblable aux constantes, variables et les fonctions pour lesquelles l'utilisateur peut définir les propriétés et les méthodes. Elle est ce qui définit notre objet.

Elle regroupe d'une part des variables qui seront nos attributs d'objet, et d'autre part des méthodes qui seront les fonctions de cet objet. Une classe est un concept, une simple définition. Un objet lui est la concrétisation de ce concept.

# Les Classes

## Bénéfices d'une classe

- L'héritage grâce auquel on accède aux propriétés d'une classe dans une autre classe
- Evite la duplication de code
- La diffusion de type qui permet à l'utilisateur d'utiliser le type de classe
- La gestion des ressources mémoires (gestion automatique )



# Les Classes

## Création de classe en Swift

*Swift nous permet de créer une classe soit dans un fichier à part soit directement dans le fichier main.swift.*

## La syntaxe :

```
class nomDeClasse  
    {  
        //Attributs / propriétés  
        //Méthodes  
    }
```

# Les Classes

## Exemple

```
class Personne
{
    var nom: String    //Attributs
    var prenom: Int
    var age: Int
}
```

# Les Classes

## Instancier une classe

```
Var objet = nomClasse ()
```

## Exemple

```
Var personne = Personne()
```

# Les Classes

## Comparer JAVA à Swift

- Contrairement à JAVA où on utilise *this* pour accéder à l'objet en cours en Swift on utilise *self*

```
self.age = 12
```

- Contrairement à java où le constructeur porte le nom de la class en Swift on appelle la fonction `init()`

```
init (age: Int)
{
    self.age = age
}
```

# Les Classes

## Comparer JAVA à Swift

- Swift accepte tout comme JAVA la surcharge de classe ainsi que l'héritage et le polymorphisme

```
class Acteur : Personne // Acteur hérite de Personne
{
var x : Int
// Polymorphisme, utilisation de la méthode init() de la
// classe Personne (superClasse ou classe mère)
override init() // le mot clé override est obligatoire
    {
        super.init (age: x) //même syntaxe que java
pour super
    }
}
```

# Les Classes

## *Les propriétés en Swift*

- **Lazy Stored Property**

Swift fournit une propriété flexible appelée *Lazy Stored Property* qui ne calculera pas la valeurs initiales lorsque la variable est initialisée pour la première fois. Le mot clé «paresseux» est utilisé avant la déclaration de la variable pour l'avoir comme une propriété mémorisée paresseuse.

### **Les propriétés sont utilisées:**

- ☐ Pour retarder la création de l'objet
- ☐ Lorsque la propriété dépend d'autres parties d'une classe et qu'elle n'est pas encore connue

# Les Classes

## *Les propriétés en Swift*

- Getter et Setter

```
var getSetterNomVariable: (Type)
{

    get {

        return self.propriete

    }
    set (propriete)
    {
        self.propriete = propriete
    }

}
```

# Les Classes

## Les propriétés en Swift

- Getter et Setter

### Remplacer :

*getPropriete()* de Java par le **getterVarNom**

et *setPropriete()* de Java par **getterVarNom = (propriete)**



# Les Structures

*Les structures en Swift rejoignent les Structures en C sauf qu'en Swift :*

*Les structures sont créées comme les classes grâce au mot clef **struct***

## *Exemple*

```
struct Individu
{
    var nom: String
    var prenom: String
}
```

# Les Structures

- Structure permet de créer un fichier unique et d'étendre son accessibilité automatiquement à d'autres blocs.
- Dans Structure, les valeurs des variables sont copiées et transmises dans les codes suivants en retournant une copie des anciennes valeurs afin que les valeurs ne puissent pas être modifiées.
- **Exercice applicatif** : Pour vérifiez les affirmations ci-dessus
- **But des Structures**
  - ☐ Encapsuler des valeurs de données simples.
  - ☐ Copier les données encapsulées et leurs propriétés associées par « valeurs » plutôt que par «références».

# Les Enumérations

## Exemple

```
enum JourSemaine
```

```
{   case Lundi  
    case Mardi   }
```

```
Var jrSemaine= JourSemaine.Lundi //Associé à une variable  
jrSemaine= .Mardi // exemple pourMerdi
```

```
switch jrSemaine  
{  
    case .Lundi:  
        print("1 er jour")  
    case .Mardi:  
        print("2 eme jour")  
}
```