



Introduction à la programmation iPhone/iPad/iWatch/iTV,... (iOS)

Découverte du langage Swift

UV AMSE

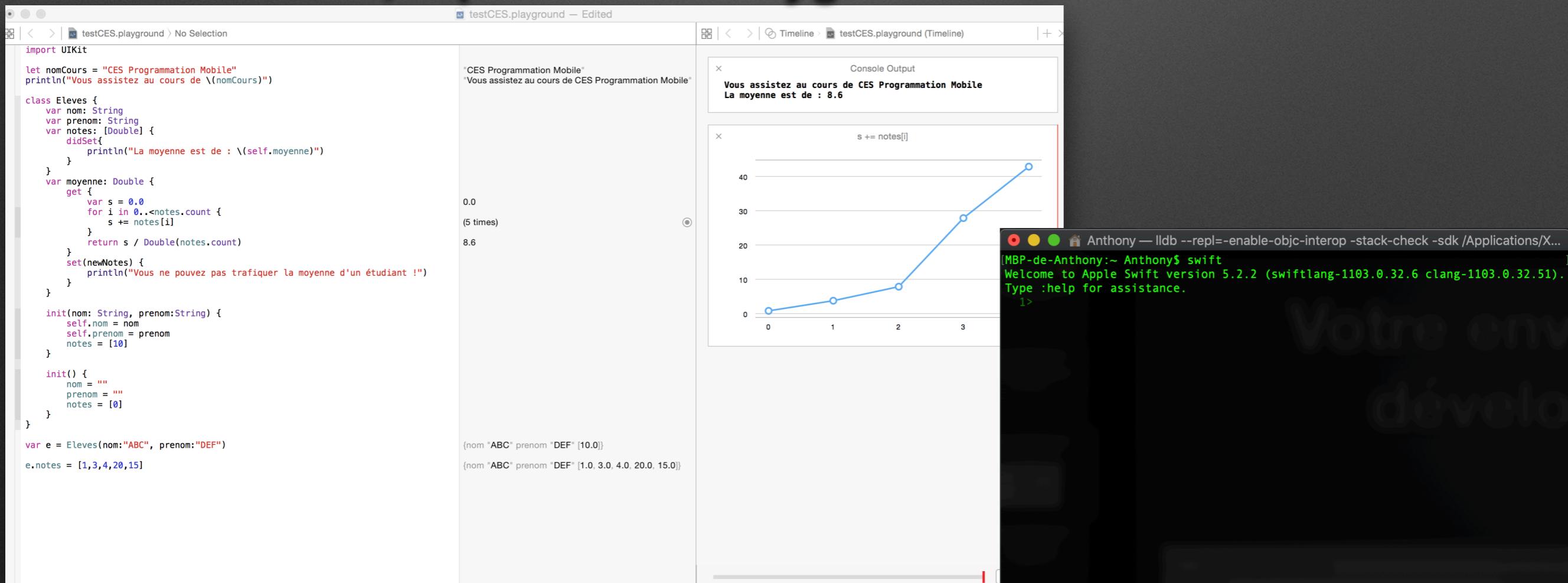
Anthony Fleury (anthony.fleury@imt-lille-douai.fr)
IMT Lille Douai

Présentation de Swift

- Swift est un langage très récent, créé par Chris Lattner, chez Apple, annoncé le 2 Juin 2014 et sorti pour sa première version stable le 19 septembre 2014. v2 du langage sortie en septembre 2015. Annonce également en 2015 de sa mise en Open-Source. v3 en septembre 2016 (avec Xcode 8) avec des changements substantiels niveau syntaxe. Encore des changements avec Swift 4 en septembre 2017. Version actuelle : Swift 5.4 (Novembre 2020)
- Un langage orienté objet, qui reprend des choses de plusieurs langages (dont le langage d'Apple - Objective C) avec une notation plus récente et plus propre
- Pour cela, création d'un réel nouveau langage par Apple (sans en faire une surcouche d'un autre langage).
- Annoncé comme plus rapide par Apple, compatible avec les « Runtimes » existants.
- Langage également disponible sur d'autres plateformes. Depuis Septembre 2020, Swift est disponible sous windows. Disponible depuis quasiment le début sous Linux.

Votre environnement de développement (1)

- Sous MacOS : XCode (récupérable gratuitement sur AppStore pour ceux qui ont) + lignes de commandes. XCode a deux modes : projet ou Playground pour écrire en mode « script » du code. Idem avec REPL en ligne de commande, équivalent à Playground.



Votre environnement de développement (2)

- En java, pour faire « Hello World », vous devez :
 - Déclarer une classe et une méthode main(),
 - Faire un System.out.println... avec un ; à la fin
- En Swift, vous avez un fichier main.swift (avec CE nom), avec comme seul contenu :

`print("Hello World")`
- Ce fichier est créé par la commande swift comme vu précédemment.

Premier programme Swift

- En Swift, vous avez un fichier `main.swift` (avec CE nom), avec comme seul contenu :
`print("Hello World")`
- Lors d'utilisation d'interface graphique, vous aurez un `AppDelegate.swift` contenant un `UIApplicationMain` mais généré automatiquement.
- Le code dans ces fichiers est le seul autorisé au premier niveau (sans être dans une fonction ou une méthode).

Déclarations, instructions, types simples

Instructions

- Par défaut en swift une instruction est sur une ligne.
- Le ; n'est pas obligatoire pour terminer une instruction. Par contre, si vous devez mettre plus d'une instruction par ligne, il est obligatoire pour terminer les instructions.
- Exemple :

```
print("Hello World")
```

```
print("ligne 1"); print("ligne 2")
```

Variables et constantes

- Les déclarations de variables se font avec le mot clé var, et les constantes avec le mot clé let.
- Le type est précisé après le caractère « : » mais peut-être omis si vous donnez une valeur (inférence de type – ATTENTION, différent d'un langage à typage dynamique)
- Exemples :
 - `let chaîneHello = « Hello » //Chaine de caractères`
 - `var chaîneCarac: String // idem`
 - `let π = 3.141592 // Double`
- Contrairement à Java on peut mettre n'importe quel caractère dans des identifiants Swift, y compris des caractères grecs, chinois, des emojis, etc. (`let 你好 = « 你好世界 », let 🐶🐮 = « ChienVache »`)

Variables et constantes :

TESTEZ !

- Si vous avez un Mac, utilisez votre propre Xcode, sinon utilisez mesh central.
- Testez ce que vous lisez, pour apprendre à écrire en Swift.

Types de base

- Les types entiers : Int sera un type de la taille de l'architecture (32 ou 64 bits). Vous avez ensuite des types entiers de toutes les tailles de 8 à 64 bits avec Int8, Int16, Int32 et Int64.
- Types entiers non signés : UInt idem à Int (non signé voulant dire que vous n'avez que des valeurs positives).
- Réels : comme en Java, vous avez Float (32 bits) et Double (64 bits). Double est le type par défaut lors d'inférence de type sur des réels.
- Booléens avec Bool (et les valeurs true et false).
- Chaines de caractères : String
- Caractères : Character. Attention, ils sont entre « « » aussi, donc pas d'inférence de type : let caracA : Character = « A »

Les opérateurs

- Les opérateurs sont les mêmes qu'en Java :
 - = pour l'affectation
 - +, -, *, / pour les 4 opérations, % pour le modulo
 - ~~++, -- pour l'incrémentation et la décrémentation (supprimé dans Swift 3)~~
 - && et || pour ET et OU
 - ! pour l'opérateur non
 - ~ pour le non bit à bit
 - == et != pour égalité et différence
 - <, >, <=, >= pour les comparaisons

L'opérateur + sur les chaînes de caractères et construction de chaînes complexes

- + va permettre de concaténer des chaînes de caractères :

```
let hello = « Hello »
```

```
let world = « World »
```

```
var message = hello + world
```

- Pour les caractères il y a la méthode append :

```
let pointEx: Character = « ! »
```

```
message.append(pointEx)
```

Les tableaux

- Les tableaux en Swift sont déclarés comme suit :

```
var tab: [type]
```

Exemple : var tableauDouble: [Double] = [1, 3, 4, 5]

- Également, de la même manière, des tableaux en plusieurs dimensions :

```
var matriceDouble: [[Double]] = [ [1, 2], [3, 4] ]
```

- Vous avez aussi des méthodes d'insertion et de suppression d'éléments :

matriceDouble.insert([5,6], at: 0) va donner la matrice [[5, 6], [1, 2], [3, 4]]

matriceDouble.remove(at: 0) va redonner le tableau de départ.

- La construction d'un tableau vide se fera par, par exemple : var tableau = [Int]()

Les valeurs optionnelles

- En java, nous pouvons écrire, pour un type donné (Eleve par exemple) :

Eleve e = new Eleve();

ou : Eleve e = null;

- Cette seconde ligne indique que « e » désignera à un moment un élève valide mais que pour l'instant il ne désigne aucun élève, aucune case mémoire.
- En swift il y a la valeur nil pour cela.
- Cependant, il est interdit d'écrire :

~~var chaineHello: String = nil~~

- En swift, lorsque l'on a un objet, il pointe OBLIGATOIREMENT sur un objet valide de ce type.

Comment alors désigner une valeur inexistante, utile lors de traitements d'erreurs par exemple ?

Les valeurs optionnelles

- Pour cela Swift a les valeurs optionnelles.
- Elles servent à dire que l'objet dans ce cas sera soit un objet valide, soit un objet invalide, donc qu'il faudra obligatoirement vérifier dans quel cas nous nous trouvons.
- Les valeurs optionnelles fonctionnent pour tous types.
- Elles sont écrites en utilisant le « ? » :

```
var doublePouvantNePasExister: Double?
```

```
doublePouvantNePasExister = 3.0
```

```
doublePouvantNePasExister = nil
```

- Lors de la déclaration de la valeur optionnelle, elle est mise à nil sauf affectation différente par le programmeur.

Les valeurs optionnelles

- Pour accéder à la valeur d'une variable optionnelle, il faut la récupérer avec un ! :

```
var valeur = doublePouvantNePasExister!
```

- Attention, si au moment de l'accès la valeur optionnelle est à nil, l'exécution s'arrête sur une erreur.
- Une autre sorte d'optionnelle permet de ne pas avoir à récupérer la valeur spécifiquement, comme suit :

```
var optionnel: Int! = 3
```

```
var valeur = optionnel
```

- Ici pas besoin de faire une extraction de la valeur, le ! dans la déclaration indique que l'on a un optionnel qui ne va être accédé comme une variable classique.
- Attention, idem si la valeur est à nil, l'exécution s'arrête automatiquement.
- La suite sur ces valeurs optionnelles sera vue dans les parties suivantes.

Instructions de contrôles et de séquencement

Contrôle de conditions en Swift

- Les conditions sont écrites quasiment de la même manière qu'en Java (mais sans le parenthésage obligatoire) :

```
if condition {  
    instructions  
}  
else if condition {  
    instructions  
}  
...  
else {  
    instructions  
}
```

- Comme en Java, condition est un booléen (Attention à ne pas remplacer le == par un = comme en Java !!)

Contrôle de conditions en Swift

- Pour les conditions, vous pouvez écrire des expressions avec une suite de `&&`, de `||` et de `!`.
- Attention à bien mettre des parenthèses si besoin de créer une condition particulière avec des enchainements de `&&`, `||`, `!`.
- Sans parenthèse la priorité des opérateurs `&&` et `||` est la même, `!` est plus prioritaire (plus précisément pour les deux premiers, ils évaluent l'expression de gauche avant l'expression de droite pour toute évaluation d'opérateurs multiples).
- En cas de priorité égales, évaluation de gauche à droite.

Contrôle de conditions : retour sur les optionnelles

- Pour extraire la valeur d'un optional, il faut utiliser l'opérateur « ! ».

```
var testopt: Int? = 3
```

```
let test: Int = testopt!
```

- Une écriture spécifique se fait avec une condition :

```
if let test = testopt {  
    print(" \(test)")  
}
```

- Dans ce cas, vérification du fait que la valeur optionnelle existe, puis si elle existe assignation de test avec cette valeur et entrée dans le if.

Les boucles

- Les boucles se font comme en Java, avec des for, des while() ou des do...while().
- Idem que pour les conditions, la syntaxe est la même, mais sans le parenthésage obligatoire.
- Pour les boucles for, par contre, Swift 3 a retiré la syntaxe for « à la Java » pour ne garder qu'un for sur des ensembles...
- Exemple :

```
for i in 0 ..< 10 {  
    print(" \(i) ")  
}
```

```
var j = 0  
while j < 10 {  
    print(" \(j) ")  
    j++  
}
```

```
var k = 0  
do {  
    print(" \(k) ")  
    k++  
} while k < 10
```

Les boucles

- S'ajoutent aux boucles for classiques les for... in.
- La notation est simple :

```
var tableau = [1,2,3]
for i in tableau {
    print(" \\"(i) ")}
```

va afficher l'ensemble des éléments d'un tableau.
- De même avec une chaîne de caractère on itérera sur chacun des caractères.

Les opérateurs d'intervalles

- S'ajoutent, aux iterations sur les collections, la possibilité d'itérer sur des intervalles :
 - L'opérateur « ... » va définir un intervalle avec bornes comprises, exemple 1...3 couvrira 1, 2 et 3.
 - L'opérateur « ..< » va couvrir un intervalle ouvert à droite, 1..<3 couvrira 1 et 2.
- Ces opérateurs peuvent alors de même être utilisés pour des boucles, comme le for in, avec une notation comme : for var i in 1...3 (en Swift 3 le var peut être mis ou non).

Retour sur les tableaux

- Les intervalles vus précédemment permettent de simplifier les notations des opérations sur les tableaux.
- Exemple :

```
var tab = [1,2,3,4,5,6]
tab[1...3] = [7, 8, 9]
```

- Il est du coup aussi possible de tester une égalité sur des parties de tableau :

```
var test1 = [1,2,3,4,5,6]
var test2 = [8,7,3,4,5,9]
test2 == test1 // FAUX
test2[2...4] == test1[2...4] // VRAI
```

- A noter que si deux tableaux ont des valeurs identiques à l'intérieur, sans être les mêmes tableaux, vous pouvez tout de même tester leur égalité avec ==, cela testera l'égalité des différents éléments des deux tableaux.

Les « one side ranges » (Swift 4)

- Avant Swift 4, il fallait donner deux bornes à des intervalles (logique)
- Dans Swift 4, le langage va tenter d'inférer une borne lorsqu'elle n'est pas donnée :

```
var planètes = ["Mercure", "Venus", "Terre", "Mars", "Jupiter",
    "Saturne", "Uranus", "Neptune"]
let enDehorsCeintureAstéroïdes = planètes[4...]
let quatrePremières = planètes[..<4]
```

- Évite d'utiliser `planètes.startIndex` et `planètes.endIndex`
- Utilisable aussi dans les `switch case` (notion vue en US 1-3) :
 - `case ...4:` tout ce qui est inférieur à 4
 - `case 2...:` tout ce qui est supérieur à 2

La notion de tuples

Les tuples

- Les tuples sont un moyen de rassembler deux valeurs de types différents en un seul type.
- Exemple :

```
let matièreAlgo: (String, Int) = (« IAP », 3) // Décrit une matière avec son nom et son coefficient.
```

- Par défaut le nom des éléments est .0 et .1. matièreAlgo.0 sera donc « IAP ». Il est possible aussi de nommer les éléments d'un tuple :

```
let matièreAlgo: (nom:String, coeff:Int) = (« IAP », 3)
```

- et accéder au nom de la matière par matièreAlgo.nom

La notion de fonctions

Les fonctions

- Contrairement à Java, vous pouvez écrire des fonctions en dehors de toute classe.
- Ces fonctions de « top level » seront donc appelées par une notation parenthésées sans notion de classe avant. Cela implique que l'on peut très bien écrire un code swift sans aucun objet à l'intérieur.
- La déclaration des fonctions diffère de la déclaration des méthodes en Java. Le type de retour est placé après et peut être multiples (avec les tuples).

Déclaration et appel d'une fonction

- Une fonction se déclare comme suit :

```
func nomFonction (parametre1: type1, ..., parametreN: typeN) -> typeRetour
```

- Si il n'y a aucun paramètre d'entrée, les parenthèses sont mises et laissées vides.
- Si il n'y a pas de paramètre de retour, le -> type n'est pas du tout mis.
- Les types de paramètres peuvent aussi être des optionnels, dans ce cas, on rajoutera le « ? ».

Exemples de fonctions

- Un exemple d'une fonction retournant un entier et recevant un entier :

```
func factoriel(n: Int) -> Int {  
    var res = 1  
    for i in 1...n {  
        res = res * i  
    }  
    return res  
}
```

- Exemple d'une fonction ne retournant rien et recevant deux valeurs :

```
func tailleIntervalle(min: Int,max: Int) {  
    print(" Début : \(min), Fin : \(max), Taille : \(max - min) ")  
}
```

- L'appel de ces fonctions se fait tout simplement en écrivant leur nom avec les valeurs des paramètres et la récupération de la valeur de retour :

```
let f15 = factoriel(15)  
print(" \(f15) ")
```

```
tailleIntervalle(2,10)
```

Fonctions à plusieurs valeurs de retour

Déclarée avec un tuple comme valeur de retour (optionnel dans ce cas) :

```
func minMax(tableau: [ Int ]) -> (min: Int, max: Int)? {  
    if tableau.isEmpty { return nil }  
    var valMin = tableau[0]  
    var valMax = tableau[0]  
    for i in tableau[1..        if i < valMin {  
            valMin = i  
        }  
        else if i > valMax {  
            valMax = i  
        }  
    }  
    return (valMin, valMax)  
}
```

L'utilisation se fait ensuite naturellement en récupérant le tuple :

```
if let bornes = minMax([8, -6, 2, 109, 3, 71]) {  
    print("Le minimum est \(bornes.min) et le maximum est \(bornes.max)")  
}
```

Noms de paramètres externes

- Swift possède une possibilité de donner ce qui est appelé des noms de paramètres externes, qui vont faire partie de la signature de la méthode.

- Exemple :

```
func concatener(chaine s1: String, avecLaChaine s2: String, etLeSeparateur sep: String) -> String {  
    return s1 + sep + s2  
}
```

Pour appeler cette méthode, nous ferons par exemple :

concatener(chaine: « Hello », avecLaChaine: « World », etLeSeparateur: « , »)

- Ces noms font partie du nom de la méthode qui est alors concatener(chaine: avecLaChaine: etLeSeparateur:)
- s1 reste le nom interne pour la première chaîne dans la méthode, chaîne sera son nom à l'extérieur.
- Vous pouvez aussi utiliser le # devant un nom interne pour donner le même nom en interne et en externe.

Les classes en Swift

Définir une classe avec ses données

- Une classe est définie par le mot clé **class**. Dans la classe vous allez pouvoir définir :
 - Des données membres,
 - Des méthodes de classe (l'équivalent de static en java)
 - Des méthodes d'instance (non static en java)
 - Des constructeurs
- Votre classe possède un membre spécifique, **self**, qui est une référence sur lui-même.

Exemple de classe

```
class Eleve {  
    var nom: String  
    var prenom: String  
    var notes: [Double]  
    var moyenne: Double  
  
    func descriptionEleve() {  
        print("Nom : \(nom), Prénom : \(prenom)")  
    }  
  
    class func nomClass() -> String {  
        return "Eleve"  
    }  
}
```

- Les données sont déclarées comme hors des classes.
- Pour les méthodes, la déclaration est la même que hors des classes, et pour les méthodes static nous utilisons le mot clé class pour définir une méthode de classe.

Les constructeurs

- Les constructeurs sont des méthodes ayant toujours le nom « init ».
- Pour la classe précédente, nous pouvons écrire le constructeur suivant :

```
init(nom: String, prenom:String) {  
    self.nom = nom  
    self.prenom = prenom  
    notes = [10]  
}
```

- Ce constructeur va recevoir un nom et un prénom et va initialiser le nom, le prénom et le tableau de note avec une seule note à 10.
- Attention cependant il va se plaindre de l'absence d'initialisation de moyenne... (cf. plus loin)
- ATTENTION, pour les constructeurs, c'est un cas spécifique, par défaut les paramètres ont un nom externe égal au nom interne.
- Pour construire un nouvel Eleve avec ce constructeur, il faudra donc écrire :

```
var e = Eleve(nom: « ABC », prenom: « DEF »)
```

- Qui va donner comme nom ABC et comme prénom DEF.

Les constructeurs (2)

- Pour éviter d'avoir à nommer les paramètres, nous pouvons demander à ne pas utiliser le nom externe avec _ :

```
init(_ nom: String, _ prenom:String) {  
    self.nom = nom  
    self.prenom = prenom  
    notes = [10]  
}
```

- Ainsi, nous pourrons écrire :

```
var e = Eleves("« ABC »", "« DEF »")
```

- Vous pouvez aussi déclarer un constructeur sans paramètre, qui sera le constructeur par défaut (sinon créé automatiquement).
- Enfin, si vous voulez appeler un constructeur dans un autre, vous devez utiliser le mot clé convenience :

```
convenience init() {  
    self.init("", "")  
}
```

- Par exemple ce constructeur sans paramètre qui va appeler le constructeur à deux paramètres avec des chaînes vides.

Les propriétés calculées

- Certaines des variables peuvent être directement calculées. Par exemple, pour notre élève, si nous avons la moyenne, lorsque l'utilisateur voudra la récupérer il faut la calculer. Cela peut être fait avec les blocs get {} et set {} de la variable moyenne.
- A la place de sa déclaration simple, nous écrivons :

```
var moyenne: Double {  
    get {  
        var s = 0.0  
        for i in 0 ..< notes.count {  
            s += notes[i]  
        }  
        return s / Double(notes.count)  
    }  
    set(newNote) {  
        print("Vous ne pouvez pas trafiquer la moyenne d'un étudiant !")  
    }  
}
```
- Lors d'une tentative d'écriture de la moyenne, la méthode set sera appelée avec la nouvelle note dans newNote.
- Lors d'une tentative de lecture de la moyenne, la méthode get sera appelée et recalculera la moyenne en fonction des notes actuelles.

Les « observateurs »

- Pour une variable donnée, vous pouvez ajouter un bloc qui sera appelé lorsque la variable sera modifiée ou en passe de l'être.
- Pour cela, vous avez les blocs didSet et willSet. Exemple avec didSet :

```
var notes: [Double] {  
    didSet{  
        print("La moyenne est de : \(self.moyenne)")  
    }  
}
```

- Avec la propriété calculée moyenne, lorsque le tableau de note sera changée, la moyenne sera recalculée et affichée.