

DEVELOPPEMENT MOBILE iOS avec Swift

Les Fonctions

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



Séquence 2: Swift

- Plan
    - Fonctions
        - Définitions
        - Fonctions et paramètres
        - Différentes types de Fonctions

Fonction: Définition

La définition décrit ce que fait la fonction, ce qu'elle s'attend à recevoir et ce qu'elle retourne une fois terminée.
La définition facilite l'appel de la fonction sans ambiguïté dans votre code:

```
print(sayHello(" UVS")) // affiche "Hello, UVS!
"
print(sayHello(« IDA")) // prints "Hello, IDA!"
```

Definition et Appel de fonctions

On peut définir le corps d'une fonction en combinant la création du message et l'instruction ***return*** en une seule ligne:

```
func sayHelloAgain(personName: String) -> String
 {
   return "Hello again, " + personName + "!"
}
 print(sayHelloAgain(« NDIAYE")) // prints "Hello again, NDIAYE!"
```

Fonctions sans Paramètre

Une fonction peut être déclarée sans paramètre.
Voici une fonction sans paramètre d'entrée, qui renvoie toujours le même message String à chaque appel:

```
func sayHelloWorld() -> String
 {
    return "hello, world"
}
 print(sayHelloWorld()) // prints "hello, world"
```

Fonctions avec des Paramètres

- Une fonction peut aussi avoir un ou plusieurs paramètres qui sont au niveau de la déclaration de la fonction séparés avec des virgules
- Exemple : cette fonction prend en paramètre le nom d'une personne pour dire Bonjour

```
func sayHello(personName: String, alreadyGreeted: Bool) -> String
 {
    if alreadyGreeted {
        return sayHelloAgain(personName)
    } else {
        return sayHello(personName)
    }
}
print(sayHello("NDIAYE", alreadyGreeted: true))
// prints "Hello again, NDIAYE!"
```

Fonctions Sans valeurs de retour

- Il n’est pas obligatoire de définir le type de retour d’une fonction.
- Dans l’exemple suivant, la fonction *sayHello (_ :) , appelée sayGoodbye (_ :), qui affiche sa propre valeur String plutôt que de la renvoyer:*

```
func sayGoodbye(personName: String)
{       print("Goodbye,
\(personName)!") }
sayGoodbye("NDIAYE")
// prints "Goodbye, NDIAYE!"
```

Fonctions: Avec de multiples valeurs de retour

- Il est possible d'utiliser ***un tuple*** comme type de retour pour qu'une fonction renvoie plusieurs valeurs.
- L'exemple suivant renvoie la plus grande et la plus petite valeurs dans un tableau de type ***int***

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

```
let bounds = minMax([8, -6, 2, 109, 3, 71])
print("min is \(bounds.min) and max is \
(bounds.max)")
// prints "min is -6 and max is 109"
```

Valeur optionnelle de retour d'un Tuple

- Si le type de tuple à retourner à partir d'une fonction peut être nul, vous pouvez utiliser un type de retour facultatif pour refléter le fait que le tuple entier peut être nul.
    - vous pouvez utiliser un type de retour de tuple facultatif pour refléter le fait que le tuple entier peut être nul: ***(Int, Int)? or (String, Int, Bool)?***
- NB: Il faut préciser aussi qu'il y'a une sur les deux instructions suivantes:
    - ***(Int, Int)? :*** le tuple en entier est optionel
    - ***(Int?, Int?):*** chaque valeur de retour est optionnelle

Valeur optionelle de retour d’un Tuple

- Donc, dans l’exemple précédent de la fonction minMax(_:); la fonction retourne un tuple contenant deux entiers de type **Int.** Cependant, la fonction n’assure une vérification si le tableau est vide ou pas.

- C’est-à-dire si le tableau passer en paramètre est vide, la fonction minMax(_:) va lancer une erreur en essayant de pointer au premier index **(array[0])**

Valeur optionelle de retour d'un Tuple

- Pour capturer l'exception si le tableau est vide, il faut utiliser un type de retour d'un tuple qui est optionnel. Au lieu de lancer une erreur, la fonction minMax(_:) va retourner **null**

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {
    if array.isEmpty { return nil }
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

Valeur optionnelle de retour d'un Tuple

- On peut utiliser « l'**optional Binding** » pour vérifier si la valeur de retour est un tuple ou null.

```
if let bounds = minMax([8, -6, 2, 109, 3,
71]) {
    print("min is \(bounds.min) and max is \
(bounds.max)")
}
// prints "min is -6 and max is 109"
```

Fonction Parameter Names

- Il existe deux types de déclarations des noms de paramètres d'une fonction: déclaration **local** et **externe.** Un nom de paramètre externe est utilisé pour définir les arguments passés à un appel de fonction.
- Un nom de paramètre local est utilisé pour l'implémentation d'une fonction

```
func someFunction(firstLocalParameterName: Int,
                  secondLocalParameterName: Int) {
// function body goes here
// firstParameterName and secondParameterName refer to
// the argument values for the first and second
parameters
}
someFunction(1, secondParameterName: 2)
```

Fonction Parameter Names: spécification d'un nom de paramètre externe

- On peut passer un paramètre déclaré à l'extérieur avant de déclarer un second paramètre local, en les séparant par un espace

```
func someFunction(externalParameterName
localParameterName: Int) {
    // function body goes here, and can use
localParameterName
    // to refer to the argument value for
that parameter
}
```

Fonction Parameter Names: spécification d'un nom de paramètre externe

- Exemple: avec la fonction sayhello(_:) qui prend en paramètre deux noms de personnes et retourne une salutation des deux personnes

```
func sayHello(to person: String, and anotherPerson:
 String) -> String {
    return "Hello \(person) and \(anotherPerson)!" }

print(sayHello(to: "NDIAYE", and: "DIOP"))
// prints "Hello NDIAYE and DIOP!"
```

Fonction Parameter Names: Omitting external parameter name

- Si vous ne souhaitez pas utiliser un nom externe pour le deuxième paramètre ou les paramètres suivants d'une fonction. Pour cela, on utilise le ***underscore(_),*** au lieu d'un nom externe explicite pour ce second paramètre

```
func someFunction(firstParameterName: Int,
                  _ secondParameterName: Int) {
// function body goes here
// firstParameterName and secondParameterName refer
// to the argument values for the first and second
parameters
}
someFunction(1, 2)
```

Exemples Fonction 1/6

```swift
func sayGoodbye(localPersonName: String) {
    print("Goodbye, \(localPersonName)!")
}
sayGoodbye("NDIAYE")
// prints "Goodbye, NDIAYE!"

func sayGoodbye2(externPersonName
localPersonName: String) {
    print("Goodbye, \(localPersonName)!")
}
sayGoodbye2(externPersonName: "DIOP")
// prints "Goodbye, DIOP!"
```

Exemples Fonction 2/6

```
// Attention : illogique ? :
func sayGoodbye3(localPersonName : String,
localPersonSurName : String) {
    print("Goodbye, \(localPersonName), \
(localPersonSurName)!")
}
sayGoodbye3("NDIAYE", localPersonSurName:
"DIOP")
// prints "Goodbye, NDIAYE, DIOP!"
// => en l'absence de param externe, on
doit utiliser l'interne (sauf pour le
premier)
```

Exemples Fonction 3/6

```
func sayGoodbye3_1(localPersonName : String,
localPersonSurName : String) {
    print("Goodbye, \(localPersonName), \
(localPersonSurName)!")
}
sayGoodbye3_1("DIOP", "NDIAYE")
// prints "error: incorrect argument label in
 call (have '_:localPersonSurName:', expected
 '_:externPersonSurName:')
//  sayGoodbye4("DIOP", localPersonSurName:
"NDIAYE")
//              ^       ~~~~~~~~~~~~~~~~~~~
//                      externPersonSurName"
```

Exemples Fonction 4/6

```
// Mais
func sayGoodbye4(localPersonName : String,
    externPersonSurName localPersonSurName : String)
{
        print("Goodbye, \(localPersonName), \
(localPersonSurName)!")
}
sayGoodbye4("DIOP", localPersonSurName: "NDIAYE")
// you get "error: incorrect argument label in call
(have '_:localPersonSurName:', expected
'_:externPersonSurName:')
// sayGoodbye4("DIOP", localPersonSurName: "NDIAYE")
//             ^       ~~~~~~~~~~~~~~~~~~
//                     externPersonSurName
 "
```

Exemples Fonction 5/6

```
// la solution logique
func sayGoodbye5(localPersonName : String,
                 externPersonSurName
localPersonSurName : String) {
    print("Goodbye, \(localPersonName), \
(localPersonSurName)!")
}
sayGoodbye5("DIOP", externPersonSurName:
"NDIAYE")
// prints "Goodbye, DIOP, NDIAYE!"
```

Exemples Fonction 6/6

```
// feignant !
func sayGoodbye6(localPersonName : String,
     _ localPersonSurName : String) {
         print("Goodbye, \(localPersonName),
 \(localPersonSurName)!")
}
sayGoodbye6("DIOP", "NDIAYE")
// prints "Goodbye, DIOP, NDIAYE!"
```

Default parameter values

- On peut définir une valeur par défaut au niveau des paramètres en assignant la valeur au niveau du paramètre après avoir préciser le type de la valeur

```
func someFunction(parameterWithDefault: Int = 12) {
    // function body goes here
    // if no arguments are passed to the function
call,
    // value of parameterWithDefault is 12
}
someFunction(6) // parameterWithDefault is 6
someFunction() // parameterWithDefault is 12
```

Varadic Parameters

- Varadic paramètre accepte 0 ou plusieurs valeurs avec un spécifique type
  - Il est utilisé pour spécifier plusieurs valeurs en paramètre quand la fonction est appelée
- Les valeurs passées en paramètres sont accessible à l'intérieur de mla fonction sous la forme d'un tableau

```
func arithmeticMean(numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5)
// returns 3.0, which is the arithmetic mean of these five
numbers
arithmeticMean(3, 8.25, 18.75)
// returns 10.0, which is the arithmetic mean of these three
numbers
```

Variables et Constantes paramètres

- Par défaut, le paramètre d’une fonction est contante
  - Changer la valeur d’un paramètre au niveau d’une fonction peut entrainer une erreur de compilation
  - Cela signifie que la valeur du paramètre ne peut pas changer dans le corps de la fonction
- Mais, il est parfois utile d'avoir une copie variable d’une valeur du paramètre à utiliser
  - Vous pouvez éviter de définir une nouvelle variable dans la fonction en spécifiant un ou plusieurs paramètres comme variables

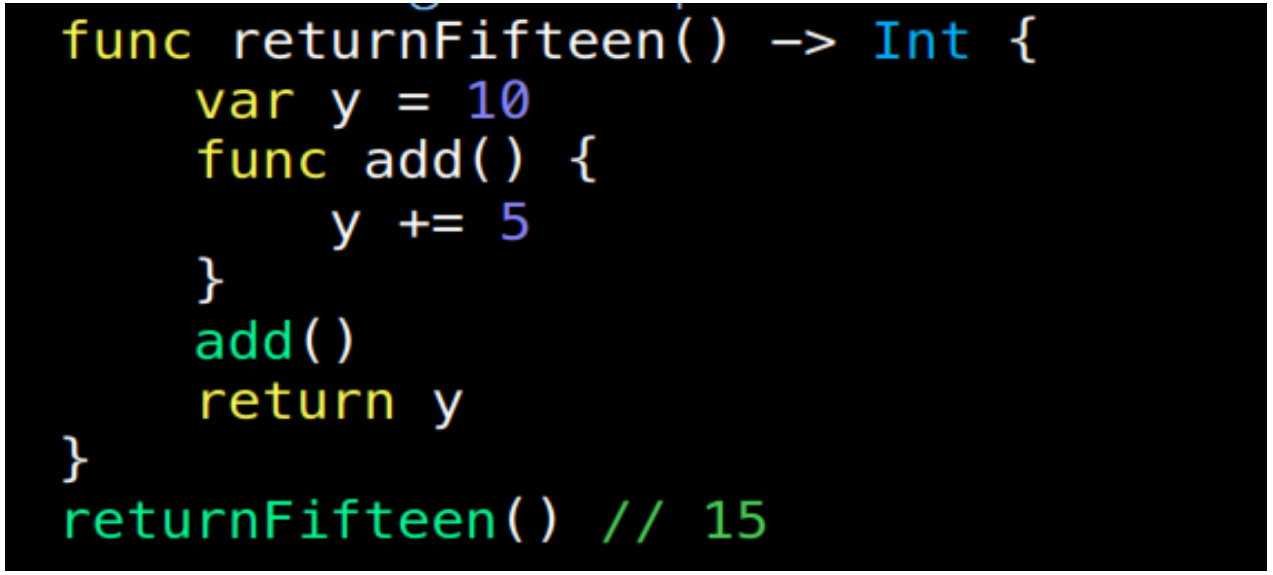
Variables et Constantes paramètres

- Définition d'une variable en paramètre en préfixant le nom du paramètre comme mot clef « ***var*** »

```
func alignRight(var string: String, totalLength: Int, pad: Character) ->
String {
    let amountToPad = totalLength - string.characters.count
    if amountToPad < 1 {
        return string
    }
    let padString = String(pad)
    for _ in 1...amountToPad {
        string = padString + string
    }
    return string
}
let originalString = "hello"
let paddedString = alignRight(originalString, totalLength: 10, pad: "-")
// paddedString is equal to "-----hello"
// originalString is still equal to « hello"!!!
```

Fonctions Imbriquées

- Une fonction peut être imbriquée
- Les fonctions imbriquées ont accès à des variables et sont déclarés dans une fonction externe. Vous pouvez utiliser fonctions imbriquées pour organiser le code dans une fonction longue ou complexe.



Fonctions Imbriquées

Les fonctions sont aussi des types de première classe: une fonction peut renvoyer une autre fonction comme valeur

```
func makeIncrementer() -> ((Int) -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}
var increment = makeIncrementer()
increment(7) // 8
```

Fonctions Imbriquées

Une fonction peut prendre en argument une autre fonction

```
func hasAnyMatches(list: [Int], condition: (Int) ->
Bool) -> Bool {
    for item in list {
        if condition(item) {
            return true
        }
    }
    return false
}
func lessThanTen(number: Int) -> Bool {
    return number < 10
}
var numbers = [20, 19, 7, 12]
hasAnyMatches(numbers, condition: lessThanTen) // true
```

Fin de la séquence 2