Introduction au développement d'applications mobiles iOS

Chapitre 1 : Introduction au langage de programmation Swift : variables & opérateurs

Le développement d'applications iOS a connu son plus grand changement, en Juin 2014, lors de la conférence annuelle WWDC, où a été annoncé le nouveau langage de programmation Swift. A l'origine, les applications étaient écrites avec le langage Objective-C qui est toujours très utilisé, mais la communauté de développeurs a vite adopté Swift.

Le langage ayant évolué graduellement dans ses détails mais aussi dans son intégration avec la librairie Cocoa qui est la base du développement iOS. Swift a été conçu avec les fonctionnalités suivantes :

- **Orienté objet**
  Swift étant un langage moderne, orienté objet, il demeure aussi purement orienté objet c'est à dire tout est objet dans le langage

- **Clarté**
  Swift est faite à écrire mais aussi à lire avec sa syntaxe qui est claire, consistante et explicite.

- **Sûre**
  Étant fortement typé, Swift s'assure de la connaissance du type de chaque référence d'un objet à tout moment

- **Econome**
  Swift étant un petit langage avec des fonctionnalités basiques, toute fonctionnalité doit être développée, ou venant du code source des librairies tierces, tels que Cocoa

- **Gestion de la mémoire**
  La mémoire est gérée automatiquement, c'est à dire que vous aurez rarement à vous occuper de tout ce qui est allocation ou gestion de la mémoire comme on le faisait avec Objective-C.

- **Compatibilité avec Cocoa**
  Les API Cocoa étant écrits principalement avec du C et de l'Objective-C, Swift a été conçu pour fonctionner (interfacer) correctement avec l'essentiel de la librairie Cocoa.

Ces fonctionnalités font de Swift un excellent langage pour apprendre à programmer sur iOS.

I. La conception du langage

Il est intéressant de savoir comment le langage Swift a été conçu et à quoi ressemble un programme écrit avec Swift.

Une commande complète dans Swift est une instruction et un fichier source a pour extension **.swift** et pouvant contenir plusieurs lignes. Un agencement typique d'un programme écrit en Swift est une instruction par ligne :

```
print("Bonjour")
print("Toi")
```

Plusieurs instructions peuvent être combinées dans une seule ligne mais il faudra les séparer par un "point-virgule".

```
print("Bonjour"); print("Toi")
```

Le point-virgule n'est pas obligatoire à la fin d'une instruction comme dans les autres langages.

Toute chose venant après deux slashes "//" d'une ligne est considérée comme un commentaire, alors Swift va l'ignorer.

```
print("quelque chose") // Ceci est un commentaire
```

Plusieurs structures dans Swift utilisent les accolades comme délimiteur :

```
class Eleve {
    func etudier() {
        print("j'apprends sagement mon cours")
    }
}
```

Pour soucis de lisibilité et de clarté, les accolades ouvrantes sont toujours suivies par un retour à la ligne et une indentation, mais rien n'empêche d'écrire le tout sur une seule ligne.

Swift est un langage compilé. C'est à dire le code doit être compilé en langage machine avant qu'il ne puisse être exécuté. Le compilateur de Swift est trop strict puisque certaines erreurs vont être rapportées à la compilation.

II. Les bases de Swift

A. Les variables et les types

Une variable est un nom attribué un objet. Techniquement, il fait référence à un objet. L'objet auquel la variable fait référence est ce qu'on appelle sa **valeur**.

Dans Swift, toute variable doit être déclaré. La déclaration d'une variable se fait avec deux mots clé : `let` et `var`. Dans Swift, une déclaration est souvent accompagnée d'une initialisation, on utilise le signe "=" pour affecter une valeur à une variable. Voici deux déclarations (et initialisations) :

```
let un = 1
var deux = 2
```

Une fois que le nom d'une variable existe, on est libre de l'utiliser. Par exemple, on peut changer la valeur de `deux` pour avoir la même valeur qu'un.

```
let un = 1
var deux = 2
deux = un
```

La dernière ligne du code utilise les deux variables déclarées dans les deux premières lignes : la variable un dans la partie droite du signe `"="` est utilisé pour faire référence à la valeur qu'elle contient c'est à dire "1"; mais la variable deux, dans la partie gauche du signe "=" est utilisée pour remplacer la valeur qu'elle contient. Une telle instruction, c'est à dire avec une variable à gauche d'un signe `"="`, est appelée une assignation et le signe `"="` est l'opérateur d'assignation. Le signe `"="` n'est pas utilisé pour tester l'égalité comme ce qu'on a dans une formule algébrique.

Les deux types de variables diffèrent du fait que la valeur variable déclarée avec le mot clé `let` ne peut pas être modifiée. Une variable déclarée avec `let` est appelée constante. Sa valeur une fois assignée ne change plus. Par exemple ce code ne sera pas compilé :

```
let un = 1
var deux = 2
un = deux //erreur de compilation
```

Il est toujours possible de déclarer une variable avec `var` pour plus de flexibilité, mais si on n'a pas l'intention de changer la valeur initiale d'une variable, mieux vaut la déclarer avec `let`.

Ce qu'il faut retenir :
Une variable est déclarée avec `let` ou `var` :

- Avec `let`, la variable devient une constante, sa valeur ne peut jamais être changée après la première assignation d'une valeur.

- Avec `var`, la variable est une vraie “variable”, et sa valeur peut -être changée dans les prochaines assignations.

Comme énoncé plus haut, Swift est un langage fortement typé. C’est à dire que tout objet a un type. Le type d’une variable peut être défini implicitement ou explicitement.

1. Déclaration explicite du type d’une variable

Pour déclarer explicitement le type d’une variable, il faut ajouter le nom du type après le nom de la variable suivi du signe par deux points :

```
var x : Int // déclare une variable de type Int sans l'initialiser
```

2. Déclaration implicite du type d’une variable

Si une variable est initialisée, sans déclarer son type explicitement, Swift déduit son type par la valeur qui lui est attribuée à l’initialisation.

```
var x = 1 // et x est maintenant un Int
```

Il est aussi possible et légal de déclarer une variable avec un type explicitement et lui assigner une valeur initiale comme dans l’exemple suivant :

```
var x : Int = 1
```

B. Variables auto-calculées

Les variables vues plus haut, ont toutes été déclarées avec une valeur fixe stocké en mémoire. Cependant, la valeur d’une variable peut aussi être auto-calculée. Ce qui veut dire qu’au lieu d’avoir une valeur fixe, la variable a la valeur de retour d’une fonction comme **`getter`**, c’est à dire quand on récupère la valeur contenue dans la variable et une fonction setter pour définir la nouvelle valeur. Exemple :

```
import Foundation

var aujourdhui : String {
    get {
        return Date().description
    }
    set {
        print(newValue)
    }
}

print(aujourdhui) //2019-10-27 20:18:38 +0000
aujourdhui = "hello" //hello
print(aujourdhui) //2019-10-27 20:18:38 +0000
```

Une variable auto-calculée doit obligatoirement être déclarée avec `var` (pas `let`). Son type doit aussi être déclaré explicitement. Le type est suivi immédiatement par les accolades. La fonction (nous allons voir les fonctions à la suite de ce cours) *getter* est appelée `get`. La fonction ***setter*** aussi est appelé `set` et n'a pas une définition formelle comme nous le verrons plus tard dans la fonction. Par contre, il y'a un paramètre `newValue` qui vient avec le setter et qui contient la nouvelle valeur que l'on veut assigner à la variable.

C. Les types prédéfinis

Chaque variable, et chaque valeur doivent avoir un type. Ainsi voici une liste des types prédéfinis dans le langage, que nous allons utiliser le plus :

- **Les Booléens (Bool)**

Les objets de type Bool n'ont que deux valeurs, communément vu comme vrai (`true`) et faux (`false`). Par exemple

```
var selected : Bool = false
```

Dans ce code `selected` est une variable de type `Bool` initialisée à `false`. Contrairement à d'autres langage comme le langage C où 0 est évalué à false et 1 à true, dans Swift rien n'est évalué à `false` sauf `false` lui-même et rien n'est non plus évaluée à `true` sauf `true` lui-même. Ainsi les objets de type Bool sont sujet aux opérations suivantes :

- `!` *(non)*

L'opérateur unaire ! donne le contraire de la vraie valeur à laquelle il est appliqué. Par exemple, si `ok` est évalué à `true`, `!ok` est évalué à `false` et vice-versa.

- `&&` *(et-logique)*

Cet opérateur retourne `true` si les deux opérandes sont évalués à `true`, sinon ça retourne `false`. Si le premier opérande est évalué à `false`, le second n'est pas évalué. Par exemple :

```
var estVrai : Bool = true
var estFaux : Bool = true
var resultat = (estVrai && estFaux)
print(resultat) //true
```

- `||` ***(ou logique)***

Cet opérateur retourne true, si au moins l'un des deux opérandes est évalué à true. Comme pour le et-logique, si le premier opérande est évalué est true, le second n'est pas évalué. Voici un exemple d'utilisation de cet opérateur :

```
var estVrai : Bool = true
var estFaux : Bool = false
var resultat = (estVrai || estFaux)
print(resultat) //true
```

Si une opération logique est compliquée, les parenthèses autour des sous-expressions peuvent aider à clarifier la logique and l'ordre des opérations.

Une situation fréquente est que nous avons une variable de type Bool et que nous voulons lui assigner l'opposé de sa valeur, c'est à dire lui assigner ***true*** si sa valeur est false et lui assigner false si sa valeur est ***true***. L'opérateur unaire ! règle le problème comme dans l'exemple suivant :

```
var unBool : Bool = false
unBool = !unBool
print(unBool) // true
```

- **Les nombres**

Les principaux types numériques sont `Int` et `Double`.
Le type `Int` représente les entiers entre `Int.max` et `Int.min` de façon inclusive. Ces valeurs min et max dépendent de la plateforme et de l'architecture dans lesquelles l'app tourne. Dans une architecture 64-bit sa valeur varie entre 2^{63}-1 et -2^{63}-1.
La façon la plus simple de représenter une variable de type Int est une valeur numérique sans sa partie décimale. Par exemple `let myInt = 12`. L'utilisation du caractère *Under score* (`_`) est autorisée pour séparer les milliers. L'utilisation de zéro au début est aussi autorisée.
Les variables de type Int peuvent aussi être représentées en utilisant le binaire, octal ou hexadécimal. Pour ce faire il faut préfixer le nombre par `0b`, `0o` ou `0x` respectivement. Ainsi, par exemple, `0x10` est l'entier `16`.

Les objets de types `Double` représentent les nombres à virgule flottante (nombre réel) avec une précision de 15 chiffres après la virgule pour une architecture 64-bit. Tout nombre avec une virgule est déduit comme un Double par défaut, ainsi l'utilisation du caractère Under score et les zéros antéposés est autorisée. Un Double peut aussi être écrit en notation scientifique comme `3e2` ainsi, tout ce qui vient après `e` est une puissance de 10. Alors `3e2` vaut `3*102=300`.

Il y'a des propriétés statiques du type `Double` à savoir `Double.infinity` et `Double.pi`, et une propriété `isZero` entre autres.

La conversion entre valeur numérique est possible par exemple pour convertir un Int en Double, il faut instancier un `Double` avec un `Int` entre parenthèse :

```
let v : Int = 10
let x = Double(v)
print(x) // 10.0, un Double
```

On peut faire pareil pour la conversion d'un `Double` en `Int`, ici tout ce qui vient après la virgule est tronqué.

Les opérateurs arithmétiques sur les types numériques

Les opérateurs arithmétiques sont comme ce qu'on a dans les autres langages.

+ (addition)
- (soustraction)
* (multiplication)
/ (division): Comme dans le langage C, la division d'un **Int** par un **Int** est un **Int**, par exemple `10/3 = 3` et non `3 + 1/3`
% (reste de la division)
Il y'a aussi l'opérateur d'incrément qui permet d'ajouter une valeur à une variable donnée et stocker le résultat dans la même variable. Pour l'utiliser, il faut systématiquement déclarer la variable par var :

```
var j = 10
j += 4
print(j) // la console affiche 14
```

Les opérateurs de comparaison

Les nombres sont comparés en utilisant les opérateurs de comparaison, qui retournent une valeur de type Bool. Par exemple, l'expression i==j teste si i et j sont égaux ; si i et j sont des nombres, "égal" signifie numériquement égal. Donc j == i est vrai seulement si i et j sont le même nombre.
Les opérateurs de comparaison sont :

- *== (opérateur d'égalité)*
- *! = (opérateur d'inégalité)*
- *< (inférieur strictement)*
- *<= (inférieur ou égal)*
- *< (supérieur strictement)*
- *>= (supérieur ou égal).*

If faut garder en tête que l'égalité des Double n'est pas toujours comme on l'espère. Pour tester si deux Double sont effectivement égaux, il est plus prudent de comparer la difference par un nombre très petit (epsilon) par exemple :

```
let estEgal = abs(x-y) < 0.00001 // abs, fonction qui donne la valeur absolue
```

- **Les String ou chaine de caractère**

Les objets de type String est une structure qui représente du texte. La façon simple de représenter un String est de délimiter par des guillemets :

```
let salut = "bonjour"
```

Le type `String` est construit comme étant de l'Unicode, donc tout caractère unicode peut être utilisé.
Le backslash est utilisé pour échapper (ou inclure) certains caractères spéciaux dans une chaîne de caractère comme : \n (retour à la ligne), \t (tabulation), \" (guillemet) \\(le backslash).

L'interpolation permet d'inclure toute valeur affichable par *print* dans une chaine de caractère même si elle n'est pas de type String. Pour ce faire, il faut utiliser le caractère d'échappement suivi de l'expression ou valeur entre parenthèse. Par exemple :

```
let n = 10
let s = "J'ai \(n) comme note à l'examen"
print(s) // J'ai 10 comme note à l'examen
```

Pour concaténer deux chaînes de caractères, il faut utiliser l'opérateur +

```
let string1 = "hello"
let string2 = " world"
let resultat = string1 + string2
print(resultat) // hello world
```

L'opérateur += aussi peut être utilisé pour concaténer deux chaînes de caractères comme dans l'exemple suivant :

```
var string1 = "hello"
let string2 = " world"
string1 += string2
print(string1) // hello world
```

Une alternative à += est la méthode d'instance append (_:) comme suit :

```
var s1 = "hello"
var s2 = " world"

s1.append(s2)
print(s1)//hello world
```

Une autre approche de concaténation des chaine de caractère est la méthode joined(separator:), c'est à dire transformer un tableau (qu'on va voir plus tard) en une chaine de caractère comme dans l'exemple suivant:

```
let s = "hello"
let s2 = "world"
let space = " "

let greeting = [s,s2].joined(separator:space)
print(greeting) //"hello world\n"
```

L'opérateur de comparaison d'égalité peut aussi être utilisé dans les chaînes de caractères. Deux variables de type String sont égales, si elles ont le même texte. Un String est plus petit qu'un autre s'il est alphabétiquement inférieur.

Il y'a des méthodes et propriétés utiles pour les opérations courantes dans les String, telles que : isEmpty qui renvoie un Bool si le String est vide c'est à dire égal est (`""`). `hasPrefix(_:)` et `hasSuffix(_:)` qui nous renseigne si une chaîne est en début ou en fin de chaîne. Les méthodes uppercased et lowercased nous donnent les

versions tout en majuscule ou en minuscule du String auquel elles sont appliquées. Par exemple :

```
var hello = "hello"
print(hello.uppercased()) // "HELLO"
```

- ## Les intervalles avec le type Range

Le type Range représente le plus souvent un intervalle entre deux nombres. Il y'a deux opérateurs (opérateur Range) qui permettent de créer un intervalle :

- … (Opérateur d'un intervalle fermé)

La notation ***a***….***b*** signifie "tout élément à partir ***a*** jusqu'à ***b*** et ***b*** y compris"

- ..< (opérateur d'un intervalle semi-ouvert)

La notation ***a***.. <***b*** signifie "tout élément à partir de a mais n'incluant pas b".
Les espaces sont autorisées autour des opérateurs Range.

Exemple :

```
let t = 1...3
```

Les intervalles Range sont d'habitude utilisés pour faire une boucle sur des nombres. Par exemple :

```
for ix in 1...4{
  print(ix) //1, puis 2, puis 3 et enfin 4
}
```

Cependant, il n'est pas possible de faire un intervalle de type range avec le premier entier plus grand que le dernier. Pour faire une boucle inverse, il faudra utiliser la fonction reversed pour parcourir l'intervalle du plus grand élément au plus petit comme sur l'exemple suivant :

```
for ix in (1...4).reversed(){
  print(ix) //4, puis 3, puis 2 et enfin 1
}
```

Il y'a la méthode contains(_:) qui permet de tester si un entier est dans un intervalle donné:

```
let k = 10;
if(5...12).contains(k) {
 print("\(k) est bien dans l'intervall")//10 est bien dans l'intervalle
}
```

Les bouts des intervalles peuvent aussi être des Double.

- ## Les Tuples

Un Tuple est une collection ordonnée de deux valeurs. Pour définir un tuple, il faut le mettre entre parenthèse et séparer les valeurs par une virgule.
Exemple :

```
var maTuple : (Int, String) = (1, "Un")
```

Dans cet exemple, une variable de type Tuple est définie avec comme valeur un couple d'un entier et d'une chaîne de caractère.
Cependant, les types des valeurs peuvent être inférés à la compilation du code comme dans l'exemple suivante :

```
var maTuple = (1, "deux")
print(maTuple)//(1, "deux")
```

Pour récupérer les valeurs contenues dans un tuple, la façon la plus simple est d'utiliser deux variables de même type que ses valeurs.
Exemple:

```
let i: Int
let s: String
(i, s) = (1, "deux")
//Ou plus simplement
let (k, m) = (1, "master")
```

- ## Les Optionnels (Optional)

Un optionnel est un type qui peux contenir un autre type. Ce qui fait d'un type ***Optional*** optionnel, est qu'il peut englober un autre objet, et peut ne rien contenir. Il faut comprendre un Optional comme une boite à chaussure, il peut contenir des chaussettes ou être vide.

Pour créer un optionnel contenant un objet, il faut utiliser son constructeur comme dans l'exemple suivant :

```
var monOptional = Optional("quelque chose")
print(monOptional)//Optional("quelque chose")
```

Dans cet exemple, la variable *monOptional* n'est pas un String ni un simple Optional, mais un Optional qui peut contenir que des String, ainsi on ne peut lui assigner qu'un Optional qui englobe un String.
Ceci est légal :

```
var monOptional = Optional("quelque chose")
print(monOptional)//Optional("quelque chose")
monOptional = Optional("autre chose")
print(monOptional)////Optional("autre chose")
```

Par contre ceci ne l'est pas :

```
var stringMaybe = Optional("yep")
stringMaybe = Optional(123) // erreur de compilation
```

Pour définir un optionnel, l'opérateur "?" peut-être utilisé à la suite de la déclaration du type d'une variable comme dans l'exemple suivant :

```
var unOpt : String?
```

Pour récuperer la valeur contenue dans un optionnel, on peut utiliser l'opérateur "!" pour forcer un optionnel à prendre la valeur qu'il englobe comme dans l'exemple suivant :

```
var unOpt : String? = "hello"
let unOptString: String = unOpt!

print(unOpt)//"Optional("hello")\n"
print(unOptString)//"hello\n"
```

Ici nous voyons bien que la variable unOptString est bien une chaîne de caractère. Un optionnel peut-être passé en paramètre à une fonction ou méthode et qu'il soit considéré dans la fonction comme l'object qu'il contient mais l'inverse n'est pas possible.
Exemple:

```
func optionalTester(_ s: String?){
   print(s)
}

optionalTester("Hello")//"Optional("Hello")\n"
```

Par contre l'opération suivante renvoie une erreur de compilation

```
func realStringTester(_ s: String){
   print(s)
}
let myOptional: String? = "will not work"
realStringTester(myOptional)//erreur de compilation
```

Comme un optionnel peut ne rien contenir, pour éviter une erreur lorsque le code est exécuté, il faut utiliser la construction ***guard statement*** suivante :

```
var prix = 10.0
var remiseOptionnelle: Double? = 0.3 //30%

if let remise = remiseOptionnelle {
   prix = prix * (1 - remise)
}
print(prix) //7.0
```

D. Les structures de contrôle

1. Les branchements (if / switch)

- If

La construction if est similaire à ce qu'on a dans le langage C. Il peut être résumé comme dans l'exemple suivant :

```
if condition {
 expressions
}

if condition {
 expressions
} else {
expressions
}

if condition {
 expressions
} else if condition {
 expressions
} else {
 expressions
}
```

- **switch**

Le branchement switch est une manière beaucoup plus élégante d'enchainer des *if..else if .. else.* Comme dans les autres langages, toutes les comparaisons doivent être exhaustives et le cas default peut être omis si tout éventualité est couvert. Exemple:

```
switch i {
   case 1:
       print("Un")
   case 2:
       print("2")
   default:
       print("\(i) un autre nombre!")
}
```

Une utilisation plus intéressante sera vue avec les types d'énumération enums.

2. Les boucles (for, while)

L'objectif des boucles est de répéter un bloc de code avec une simple petite différence à chaque itération. La différence réside principalement à savoir quand il faut arrêter la boucle. Ainsi Swift propose deux boucles basiques : **while** et **for**

- **La boucle while**

La boucle while se présente sous deux formes :

```
while condition {
 expression
}

repeat {
 expression
```

```
} while condition
```

La différence entre les deux formes est le test de la condition. Dans le second cas, le test est fait une fois que le bloc de code est déjà exécuté, c'est à dire que le code est au moins exécuté une fois. Par contre dans le premier cas, la condition est testée avant que le bloc de code ne soit exécuté.
Exemple :

```
var someInt = 10
while someInt > 0 {
    someInt -= 1
}
print(someInt) // 0
```

- **La boucle for**

La boucle for peut être schématisée par la structure suivante :

```
for variable in sequence {
    expressions
}
```

Une utilisation usuelle de la boucle for est le parcours d'un intervalle de type Range vu plus haut.
Exemple:

```
for let i in 1...3 {
    print(i)
}
```

Pour parcourir un tableau ou une séquence avec les indexes des éléments, il faut utiliser la méthode enumerated() des tableaux comme dans l'exemple suivant:

```
var abc = ["a", "b", "c"]
for (j, car) in abc.enumerated(){
    print(j, car) // 0 a, 1 b, 2 c
}
```

La boucle for peut aussi prendre une condition where additionnelle pour filtrer les élements à appliquer le bloc de code.

```
for i in 0...10 where i % 2 == 0 {
    print(i) // 0, 2, 4, 6, 8, 10
}
```

E. Les tableaux et dictionnaires

Swift, comme dans tous les langages de programmation modernes, dispose de collection de type **Array** et **Dictionnary**, avec un troisième, Set.

- **Array : tableau**

Un tableau est une collection ordonnée d'instances d'Object accessible par un index numéroté, où le numéro d'index commence à partir de 0. C'est à dire si un tableau contient 4 éléments, le premier a pour index 0 and le dernier 3.
Dans Swift, les éléments d'un tableau doivent tous être du même type. C'est à dire un tableau de Int ne peut contenir que des éléments de type **Int**.
Pour définir le type d'un tableau, la syntaxe à utiliser est la suivante :

```
var unTableau : Array<Int>
//-----  ou ------
var unAutreTableau : [Int]
```

Le type peut aussi être inféré comme dans l'exemple suivant :

```
var listeDeNoms = ["Jacob", "Marie", "Alpha", "Usman"]
```

Pour créer un tableau vide, on peut utiliser les constructions suivantes :

```
var monTableau = [Int]()
// Ou bien encore
var monAutreTableau : [Int] = []
```

Pour parcourir un tableau, la manière la plus simple est d'utiliser la boucle for comme énoncée dans les sections précédentes.

- **Dictionnaire**

Un dictionnaire est une collection non ordonnée de paires d'object. Dans chaque pair, le premier objet est la clé, et le second objet la valeur. Les clés sont souvent des String. Comme avec les tableaux, les objects contenus dans un dictionnaire doivent avoir le même type.

Pour initialiser un dictionnaire vide, la construction à utiliser est la suivante :

```
var d = [Cle:Valeur]()
```

Par exemple : `var d = [String:String]()` qui est un dictionnaire vide qui doivent contenir des objets de type String comme valeur ayant des clés de type String.

Le symbole ":" (deux-point) est utilisé entre chaque clé et valeur pour déclarer un dictionnaire avec des valeurs initiales comme suit :

```
var senegal = ["DK": "Dakar", "TH": "Thies", "ST": "Saint-Louis"]
```

Pour récupérer la valeur contenue dans une clé, il faut utiliser la syntaxe suivante :

```
senegal["DK"] // Dakar
```

Le tableau de l’ensemble des clés d’un dictionnaire peut être récupéré grace à la propriété keys comme dans l’exemple qui suit :

```
for r in senegal.keys {
    print(r)//"DK", "TH", "ST"
}
// Ou
var r = senegal.keys // ["DK", "TH", "ST"]
```