



# Swift : les bases

To infinity and beyond !

# Plan

- Syntaxe et principes de base
- Chaines de caractères
- Collections
- Boucles
- Conditions
- Fonctions
- Optionnels
- Clôtures

# Syntaxe et principes de base

# Hello, World

```
int main(void)
{
    printf("Hello, World\n");
    return 0;
}
```

# Hello, World

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

# Constantes et

```
let languageName: String = "Swift"
```

# Constantes et Variables

```
let languageName: String = "Swift"  
var version: Double = 3.0  
var introduced: Int = 2014  
var isAwesome: Bool = true
```

# Constantes et Variables

```
let languageName: String = "Swift"  
var version: Double = 3.0  
let introduced: Int = 2014  
let isAwesome: Bool = true
```

# Inférence du type

```
let languageName:=StString "Swift"  
var version:=Double = 3.0  
let introduced:=Int14 2014  
let isAwesome:=Boole= true
```

# Syntaxe et principes de base

- Une variable se déclare avec le préfixe var
- Une constante se déclare avec le préfixe let
  - La valeur d'une constante ne peut pas changer
- Le type d'une variable est fixe
  - Il peut être déduit de la valeur initiale affectée à une variable (inférence)
  - Il peut être précisé si la valeur initiale est ambiguë ou ne permet pas de définir le type

# Chaînes de caractères

# Chaînes de caractères

```
let helloString = "Hello"  
//Création d'une chaîne de caractère
```

```
let worldString = "World"
```

```
let completeString = helloString + " " + worldString +  
"!"  
let completeString2 = "\(\helloString) \(\worldString)!"  
//"Hello World!"
```

# Chaînes de caractères

```
let isEmptyValue = helloString.isEmpty  
// isEmptyValue est false
```

```
let helloWorldString = "Hello world"  
let firstCheck = helloWorldString.hasPrefix("Hello")  
let secondCheck = helloWorldString.hasSuffix("World")  
//firstCheck est true  
//secondCheck est false
```

# Caractères et chaînes

```
let chien: Character = "🐶"  
let exclamation: Character = "!"  
  
var instruction = "Attention au \(\chien) "  
instruction.append(exclamation)  
// "Attention au 🐶 !"
```

# Caractères et chaînes

- Une chaîne de caractères est une succession de caractères
- Des caractères ou d'autres chaînes peuvent être rajoutés à une chaîne de caractères existante
- Pour déclarer simplement une chaîne de caractère, on utilise les guillemets
- Une chaîne déclarée avec let n'est pas modifiable.

Swift

# Collections

# Collections

- Tableau
  - Collection d'éléments ordonnés et indexés
- Dictionnaire
  - Collection d'éléments associés à des clés
- Ensembles
  - Collection d'éléments uniques
- En Swift, les collections sont typées

# Tableaux et dictionnaires

```
var villes = ["Paris", "Bordeaux", "Lyon", "Marseille"]  
// tableau de String
```

```
var nbHabitants = {"Paris" : 225000, "Bordeaux" : 239000,  
"Lyon" : 491268, "Marseille" : 850636}  
// dictionnaire avec String en clé, Int en valeur
```

# Tableaux et dictionnaires

```
var villes = ["Paris", "Bordeaux", "Lyon", "Marseille"]  
// tableau de String
```

```
var nbHabitants = {"Paris" : 2_250_000, "Bordeaux" :  
239_000, "Lyon" : 491_268, "Marseille" : 850_636}  
// dictionnaire avec String en clé, Int en valeur
```

# Tableaux et dictionnaires

```
var villes: [String] = ["Paris", "Bordeaux", "Lyon", "Marseille"]  
// tableau de String
```

```
var nbHabitants: [String:Int] = ["Paris" : 2_250_000,  
"Bordeaux" : 239_000, "Lyon" : 491_268, "Marseille" : 850_636]  
// dictionnaire avec String en clé, Int en valeur
```

# Tableaux et dictionnaires

```
var villes = ["Paris", "Bordeaux", "Lyon", "Marseille"]  
// tableau de String
```

```
var nbHabitants = {"Paris" : 2_250_000, "Bordeaux" :  
239_000, "Lyon" : 491_268, "Marseille" : 850_636}  
// dictionnaire avec String en clé, Int en valeur
```

# Tableaux et dictionnaires

```
var villes = [String]()
// création d'un tableau de String vide
```

```
var nbHabitants = [String:Int]()
// création d'un dictionnaire avec String en clé, Int en
valeur vide
```

# Tableaux et dictionnaires

```
var villes = [String]()
var villes = Array<String>()
// création d'un tableau de String vide

var nbHabitants = [String:Int]()
var nbHabitants = Dictionary<String, Int>()
// création d'un dictionnaire avec String en clé, Int en
valeur vide
```

# Tableaux et dictionnaires

```
var villes = [UnType]()
var villes = Array<UnType>()
// création d'un tableau de UnType vide

var nbHabitants = [TypeClé:TypeValeur]()
var nbHabitants = Dictionary<TypeClé, TypeValeur>()
// création d'un dictionnaire avec TypeClé en clé,
TypeValeur en valeur vide
```

# Tableaux

```
villes += ["Toulouse"]
villes.append("Toulouse")
// on ajoute "Toulouse" à la fin du tableau

villes.insert("Toulouse", at: 2)
// on ajoute "Toulouse" à l'index 2 du tableau

villes.remove(at: 5)
// on retire l'élément à l'index 5 du tableau

villes.removeLast()
// on retire le dernier élément du tableau
```

# Tableaux

```
let uneVille = villes[4]
// on récupère l'élément à l'index 4
```

```
villes[4] = "Massilia"
// on remplace l'élément à l'index 4 par « Massilia »
```

```
villes[2...4] = ["San Francisco", "New York"]
// on remplace les éléments 2 à 4 (inclus) par les nouveaux
```

```
let nbVilles = villes.count
// on récupère le nombre d'éléments dans le tableau
```

```
let vide = villes.isEmpty
// on récupère un booléen selon si le tableau est vide
```

# Dictionnaires

```
nbHabitants["Toulouse"] = 447_340
// on ajoute la valeur 447_340 à la clé "Toulouse"
```

```
nbHabitants["Toulouse"] = 447_341
// on met à jour la valeur à la clé "Toulouse"
```

```
nbHabitants["Toulouse"] = nil
nbHabitants.removeValue(forKey: "Toulouse")
// on retire la valeur pour la clé "Toulouse"
```

# Dictionnaires

```
let keys = [String](nbHabitants.keys)
// on récupère les clés du dictionnaire dans un tableau
```

```
let values = [Int](nbHabitants.values)
// on récupère les valeurs du dictionnaire dans un tableau
```

```
let nbElements = nbHabitants.count
// on récupère le nombre d'éléments dans le dictionnaire
```

```
let vide = nbHabitants.isEmpty
// on récupère un booléen selon si le dictionnaire est vide
```

# Ensembles

```
var cities = Set<String>()
//Création d'un ensemble vide (pas de syntaxe "simplifiée")

cities.insert("Paris")
cities.insert("Bordeaux")
//Insertion

var cities2: Set = ["Bordeaux", "Lyon", "Marseille"]
//Création d'un ensemble à partir de la syntaxe simplifiée
des tableaux
```

# Ensembles

```
cities.contains("Paris")
// Le résultat est true
```

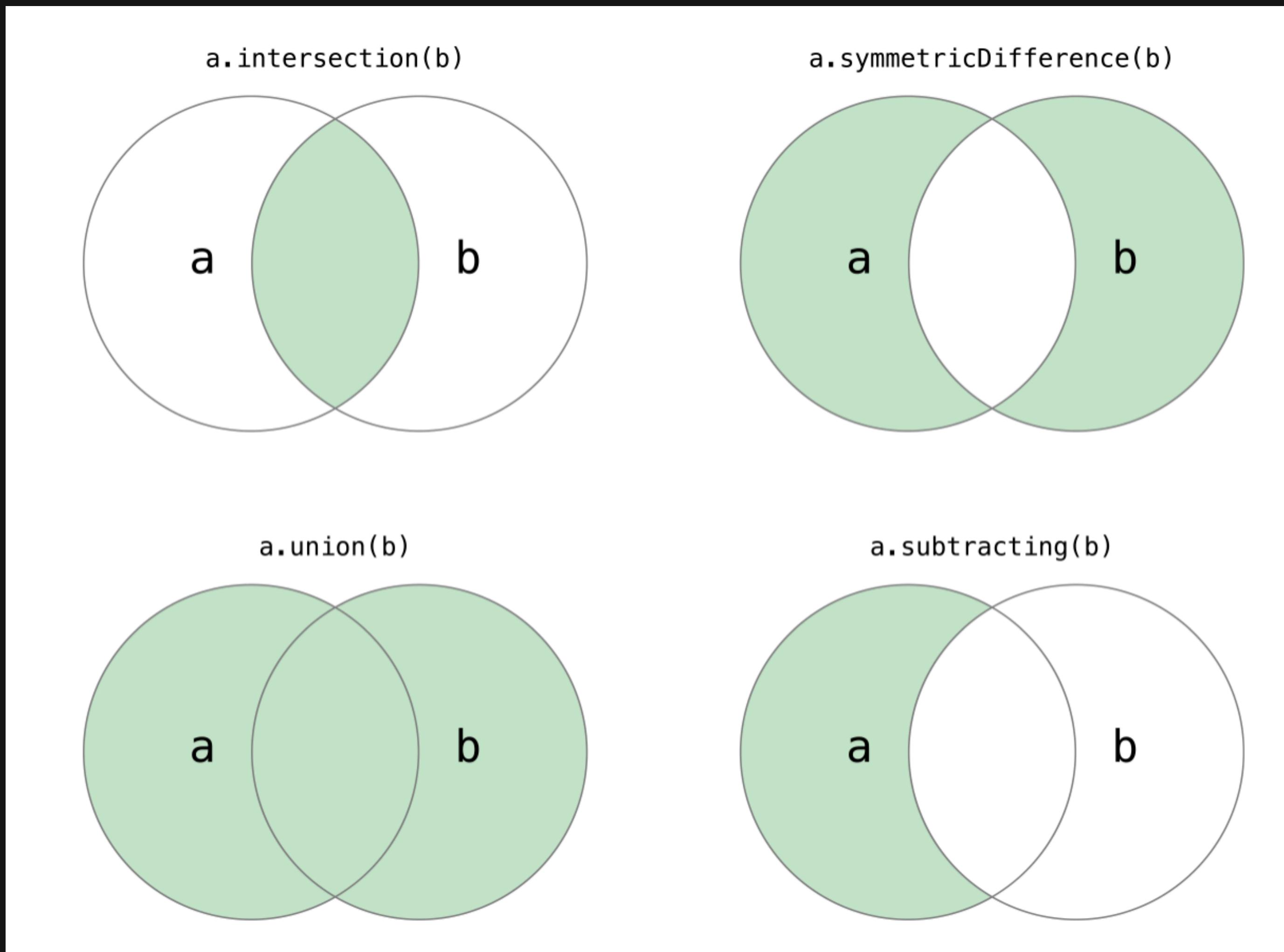
```
cities.intersect(cities2)
// {"Bordeaux"}
```

```
cities.union(cities2)
// {"Bordeaux", "Marseille", "Paris", "Lyon"}
```

```
cities.subtract(cities2)
// {"Paris"}
```

```
cities.symmetricDifference(cities2)
// {"Paris", "Lyon", "Marseille"}
```

# Ensembles



Swift

# Boucles

# Boucles

- While / Repeat-While
- For-in
  - Énumération rapide des éléments dans une collection

# While / Repeat-While

```
while !swiftMaster {  
    learnSwift()  
}
```

# While / Repeat-While

```
while !swiftMaster {  
    learnSwift()  
}  
  
repeat {  
    technicalTraining()  
}  
} while !swiftMaster
```

# For-in

```
for i in 0..<10 {  
    doSomething()  
}
```

# For-in

```
for _ in 0..<10 {  
    doSomething()  
}
```

# For-in : Chaines de caractères

```
for caractère in "🎁🎁🎁🎁" {  
    print(caractère)  
}
```



# For-in : intervalle

```
for nombre in 0...4 {  
    print("\\"(nombre) " x 5 = \"(nombre*5)")  
}
```

```
0 x 5 = 0  
1 x 5 = 5  
2 x 5 = 10  
3 x 5 = 15  
4 x 5 = 20
```

# For-in : tableau / ensemble

```
for ville in villes {  
    print("Bienvenue à "+ville)  
}
```

Bienvenue à Toulouse  
Bienvenue à Paris  
Bienvenue à Bordeaux  
Bienvenue à Lyon  
Bienvenue à Massilia

# For-in : dictionnaire

```
for (ville,nbHab) in nbHabitants {  
    print("La ville de \\"+ville+" compte "+nbHab)  
habitants")  
}
```

```
La ville de Bordeaux compte 239000 habitants  
La ville de Marseille compte 850636 habitants  
La ville de Paris compte 2250000 habitants  
La ville de Lyon compte 491268 habitants
```

# Conditions

## Conditions

If

```
ifisOk {  
    //do something  
}  
  
elseIf somethingElse {  
    //do something else  
}  
  
else {  
    //do something else  
}
```

# If

- If / Else
  - Parenthèses autour des conditions non obligatoires
  - Accolades obligatoires

# Switch

```
var greeting = "Good Morning"

switch greeting {

    case "Good Morning":
        print("It's morning")

    case "Good Evening":
        print("It's evening")

    default:
        print("I don't know")

}
```

# Switch

- Pas limité au Int / enums
- Doit être exhaustif ou inclure un cas default
- Pas de passage de cas en cas automatique
  - Pas besoin de break
  - Possibilité de retrouver le comportement "C" avec fallthrough
- Très puissant avec du pattern matching

# Guard

```
func somethingWithAge(age: Int) {  
    if age > 0 {  
        if age < maxAge {  
            //Do the right thing.  
        } else {  
            print("There is a problem: too old")  
        }  
    } else {  
        print("There is a problem: too young")  
    }  
}
```

# Guard

```
func somethingWithAge(age: Int) {  
    if age < 0 {  
        print("There is a problem: too young")  
        return  
    }  
  
    if age > maxAge {  
        print("There is a problem: too old")  
        return  
    }  
    //Do the right thing.  
}
```

# Guard

```
func somethingWithAge(age: Int) {  
    guard age > 0 else {  
        print("There is a problem: too young")  
        return  
    }  
  
    guard age < maxAge else {  
        print("There is a problem: too old")  
        return  
    }  
  
    //Do the right thing.  
}
```

# Guard

- Pattern de sortie anticipée
  - Permet de poser des conditions en début de scope
  - Évite les imbrications de if
  - Bloc else obligatoire
    - Ce bloc doit faire quitter le scope

# Fonctions

# Fonction simple

```
func learn(){  
    print("I love Swift !")  
}
```

```
learn()
```

I love Swift !

# Fonction avec paramètre

```
func learn(language: String) {  
    print("I love \(language) !")  
}
```

```
learn(language: "Objective-C")
```

I love Objective-C !

# Fonction avec paramètres

```
func learn(language: String, time: Int) {  
    print("I love \(language) \(time) times more than C#")  
}
```

```
learn(language: "Swift", time: 42)
```

I love Swift 42 times more than C#

# Fonction avec paramètres

- On peut passer des paramètres à nos fonctions
  - Les paramètres se passent entre parenthèses
  - Chaque paramètre est composé d'un nom et d'un type

# Nommage des paramètres

```
? ?  
changeColor(0.5, 0.5, 0.5, 0.5)  
? ?
```

# Nommage des paramètres

```
?           ?  
changeColor(red: 0.5, green: 0.5, blue: 0.5, alpha: 0.5)  
?
```

# Nommage des paramètres

- En Swift, on essaye de nommer ses fonctions en suivant quelques règles :
  - Clarté lors de l'utilisation : une fonction va être utilisée régulièrement. Il faut la designer de façon à ce qu'on comprenne ce qu'elle fait à la première lecture.
  - La clarté est plus importante que la brièveté : Bien que le code Swift puisse être compact, ça n'est pas un but en soit, et cela ne doit pas nuire à la clarté.

# Nommage des paramètres

- Pour une fonction, Swift donne des noms internes et externes aux paramètres.
- Possibilité de modifier le comportement en préfixant le nom lors de la déclaration : modifier le nom externe, ou le supprimer avec \_

# Nommage des paramètres

```
func changeColor(red:Float, green:Float, blue:Float,  
alpha:Float){  
    //do something  
}  
  
changeColor(red: 0.5, green: 0.5, blue: 0.5, alpha: 0.5)
```

# Nommage des paramètres

```
func changeColor(red:Float, _ green:Float, blue:Float,  
andAlpha alpha:Float){  
    //do something  
}  
  
changeColor(red: 0.5, 0.5, blue: 0.5, andAlpha: 0.5)
```

# Fonction avec paramètres par défaut

```
func learn(language: String = "Swift", time: Int = 10) {  
    print("I love \(language) \(time) times more than C#")  
}  
  
learn()
```

```
I love Swift 10 times more than C#
```

# Fonction avec paramètres par défaut

```
func learn(language: String = "Swift", time: Int = 10) {  
    print("I love \(language) \(time) times more than C#")  
}  
  
learn()  
learn(language: "Objective-C")
```

```
I love Swift 10 times more than C#  
I love Objective-C 10 times more than C#
```

# Fonction avec paramètres par défaut

```
func learn(language: String = "Swift", time: Int = 10) {  
    print("I love \(language) \(time) times more than C#")  
}  
  
learn()  
learn(language: "Objective-C")  
learn(language: "Objective-C", time:42)
```

```
I love Swift 10 times more than C#  
I love Objective-C 10 times more than C#  
I love Objective-C 42 times more than C#
```

# Fonction avec paramètres par défaut

- Il est préférable de placer les paramètres avec valeurs par défaut à la fin de la liste des paramètres (pour limiter le déplacement des paramètres dans les appels de fonctions, et améliorer la lisibilité)

# Retours

```
func learn(language: String = "Swift", time: Int = 10) -> String
{
    return "I love \(language) \(time) times more than C#"
}

let declaration = learn(language: "Objective-C", time:42)
print(declaration)
```

I love Objective-C 42 times more than C#

# Retours

```
func learn(language: String = "Swift", time: Int = 10) -> String
{
    return "I love \(language) \(time) times more than C#"
}

let declaration = learn(language: "Objective-C", time:42)
print(declaration)
```

```
I love Objective-C 42 times more than C#
```

# Retours

- Une fonction peut retourner une valeur
  - Ce retour se marque avec une `>` suivie du type retourné
  - Si la fonction ne retourne rien, on ne mets rien (équivalent à `> Void`)
  - Si un retour est prévu, il est obligatoire
  - Sinon refus de compilation

# Retours multiples

```
func loadURL() -> (Int, String) {  
    //On essaye de charger une URL  
  
    return (200, "OK")  
}
```

# Tuples

- Groupe de valeurs pouvant être retournés par une fonction
- Un tuple est typé en fonction des éléments qui le compose

```
(404, "Not Found") // (Int, String)
```

```
(0.4, 0.73, 2) // (Float, Float, Int)
```

```
(2, "aString", 3.14) // (Int, String, Float)
```

# Retours multiples

```
func loadURL() -> (Int, String) {  
    //On essaye de charger une URL  
  
    return (200, "OK")  
}
```

# Retours multiples

```
func loadURL() -> (Int, String) {  
    //On essaye de charger une URL  
  
    return (200, "OK")  
}
```

```
let status = loadURL()  
print(status)
```

```
(200, OK)
```

# Retours multiples

```
func loadURL() -> (Int, String) {  
    //On essaye de charger une URL  
  
    return (200, "OK")  
}
```

```
let status = loadURL()  
print(status.1)
```

OK

# Retours multiples

```
func loadURL() -> (Int, String) {  
    //On essaye de charger une URL  
  
    return (200, "OK")  
}  
  
let (statusCode, message) = loadURL()  
print("Réponse reçue -> \(statusCode) : \(message)")
```

Réponse reçue -> 200 : OK

# Retours multiples nommés

```
func loadURL() -> (statusCode: Int, message: String) {  
    //On essaye de charger une URL  
  
    return (200, "OK")  
}  
  
let status = loadURL()  
print(status.statusCode)
```

200

# Retours multiples

- Une fonction peut retourner un tuple de plusieurs valeurs
  - Ce tuple peut contenir des éléments nommés afin d'y accéder plus simplement
  - Sinon, on utilise l'index des éléments dans le tuple
  - On peut également décomposer un tuple dans plusieurs variables pour les exploiter

# Optionnels

# Optionnels

- Savoir si une variable contient une valeur, ou pas
- Moyen d'augmenter la sécurité et fiabilité du code
- Nécessite d'être "déballé" pour pouvoir accéder à la valeur

# Optionnels

```
let capitales = ["France" : "Paris", "USA" : "Washington",
"Canada" : "Ottawa"]
let capFrance = capitales["France"]
```

```
let intString = "42"
let intValue = Int(intString)
```

# Optionnels

```
let capitales = ["France" : "Paris", "USA" : "Washington",
"Canada" : "Ottawa"]
let capIrlande = capitales["Irlande"]
```

```
let intString = "Hello World"
let intValue = Int(intString)
```

# Optionnels

```
let capitales = ["France" : "Paris", "USA" : "Washington",
"Canada" : "Ottawa"]
let capIrlande: String? = capitales["Irlande"]
// capIrlande est un String? (chaine de caractères
optionnelle)

let intString = "Hello World"
let intValue: Int? = Int(intString)
// intValue est un Int? (entier optionnel)
```

Optionnels

# Optionnels

```
let intString = "42"  
let intValue = Int(intString)  
print(intValue)
```

Optional(42)

# Optionnels

```
let intString = "Hello World"
```

```
let intValue = Int(intString)
```

```
print(intValue)
```

????

# Optionnels

```
let intString = "Hello World"
```

```
let intValue = Int(intString)
```

```
print(intValue)
```

nil

# Optionnels

```
let intString = "42"
```

```
let intValue = Int(intString)
```

```
print(intValue)
```

```
print("La valeur entière est \(intValue)")
```

```
Optional(42)
```

```
La valeur entière est Optional(42)
```

# Optionnels

```
let intString = "42"  
  
let intValue: Int? = Int(intString)  
  
print(intValue)  
  
print("La valeur entière est \(intValue)")
```

Optional(42)

La valeur entière est Optional(42)

# Optionnels : déballage

```
let intString = "42"
```

```
let intValue: Int? = Int(intString)
```

```
print(intValue!)
```

```
print("La valeur entière est \(intValue!)")
```

```
42
```

```
La valeur entière est 42
```

# Optionnels : déballage

```
let intString = "Hello World"  
let intValue: Int? = Int(intString)  
print(intValue!)  
print("La valeur entière est \(intValue!)")
```

```
fatal error: unexpectedly found nil while unwrapping an  
Optional value
```

# Optionnels : déballage

```
let intString = "Hello World"  
  
let intValue = Int(intString)  
  
if (intValue != nil) {  
  
    print("La valeur entière est \(intValue!)")  
  
}
```

# Optionnels : déballage

```
let intString = "Hello World"  
  
if let intValue = Int(intString) {  
  
    print("La valeur entière est \(intValue)")  
  
}
```

# Optionnels : déballage

```
let intString = "42"  
  
if let intValue = Int(intString) {  
  
    print("La valeur entière est \(intValue)")  
  
}
```

# Optionnels : déballage

```
let intString = "42"

guard let intValue = Int(intString) else {

    print("Erreur")
    fatalError()
}

print("La valeur entière est \(intValue)")
```

# Optionnels

- Un optionnel est un élément qui comporte
  - Soit une valeur
  - Soit aucune valeur (=nil)
- Pour accéder à la valeur, il faut déballer l'optionnel (utiliser !)
  - Avant de déballer un optionnel, il faut s'assurer qu'il contiennent une valeur
  - Si on déballe un optionnel nil, attention au crash !
- Pour simplifier le déballage, on peut utiliser la syntaxe `if let`
  - On peut aussi utiliser `guard let` (la variable est déballée pour tout le scope)

# Clôtures

# Clôtures

- Similaires aux blocs en Objective-C
- Blocs de codes autonomes
- Peuvent être passés en paramètres ou retournés
- Les clôtures peuvent "capturer" une référence des variables de leur contexte
- Les fonctions sont des clôtures avec un nom

# Clôtures

Une fonction est une clôture avec un nom. Les clôtures, et donc les fonctions, sont des types de première classe. Une fonction est donc définie par un type bien précis, défini par ses paramètres et ses types de retours.

```
func compare (n1: Int, isBiggerThan n2: Int) -> Bool {  
    let greater = n1 > n2  
  
    return greater  
}  
  
compare(n1: 10, isBiggerThan: 30)  
//Retourne false
```

# Clôtures

```
let compare: (Int, Int) -> Bool
```

```
let compare = { (n1: Int, n2: Int) -> Bool in  
    let greater = n1 > n2  
    return greater  
}
```

```
compare(100, 20)  
//Retourne true
```

# Clôtures

```
let compare = { (n1: Int, n2: Int) -> Bool in
    let greater = n1 > n2
    return greater
}

let tab = [5, 10, 54, 2]
tab.sorted(by: compare)
//Retourne [54, 10, 5, 2]
```

sort prend en paramètre, une clôture de tri qui retourne true si son premier paramètre doit être placé avant le deuxième.

# Clôtures

```
let tab = [5, 10, 54, 2]
```

```
tab.sorted(by: {(n1: Int, n2: Int) -> Bool in
```

```
    let greater = n1 > n2
```

```
    return greater
```

```
})
```

```
//Retourne [54, 10, 5, 2]
```

# Clôtures

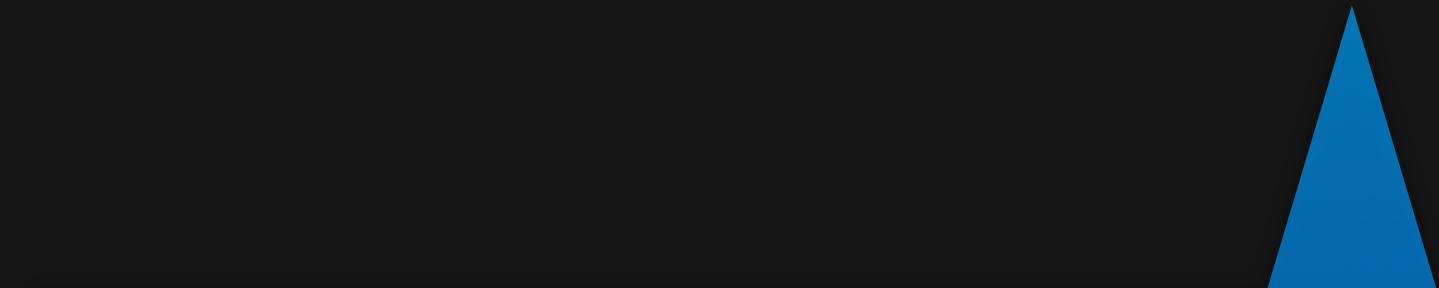
```
tab.sorted(by: {(n1: Int, n2: Int) -> Bool in  
    return n1 > n2  
})  
  
//Retourne [54, 10, 5, 2]
```

# Clôtures

```
let tab = [5, 10, 54, 2]  
tab.sorted(by: {(n1: Int, n2: Int) -> Bool in return n1 > n2 })  
//Retourne [54, 10, 5, 2]
```

# Clôtures

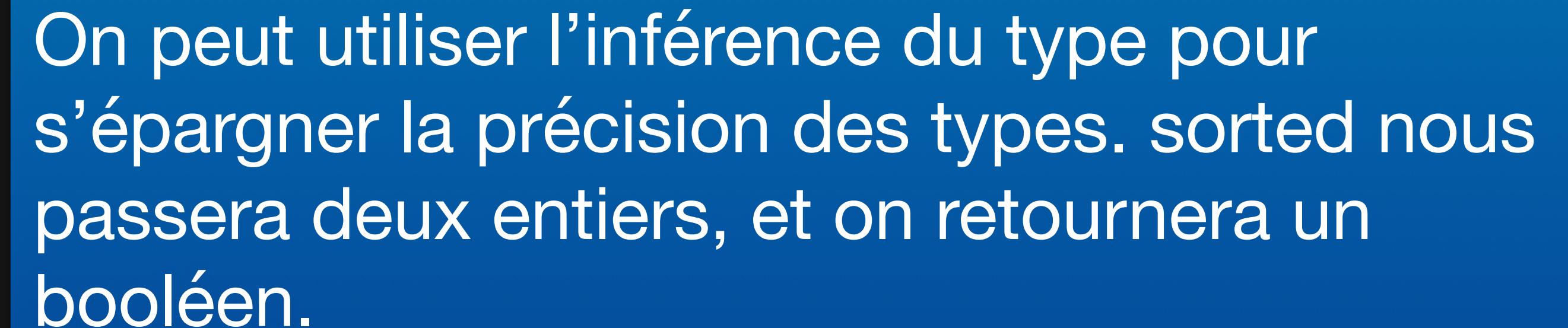
```
let tab = [5, 10, 54, 2]  
tab.sorted(by: {(n1: Int, n2: Int) -> Bool in n1 > n2 })  
//Retourne [54, 10, 5, 2]
```



Les clôtures avec une seule instruction peuvent retourner automatiquement le résultat de cette instruction.

# Clôtures

```
let tab = [5, 10, 54, 2]  
tab.sorted(by: {(n1, n2) in n1 > n2 })  
//Retourne [54, 10, 5, 2]
```



On peut utiliser l'inférence du type pour s'épargner la précision des types. sorted nous passera deux entiers, et on retournera un booléen.

# Clôtures

```
let tab = [5, 10, 54, 2]  
tab.sorted(by: {$0 > $1})  
//Retourne [54, 10, 5, 2]
```

Dans les clôtures, Swift nous donne accès aux arguments via une notation "simplifiée". \$0 pour le premier argument, puis \$1, etc.

On peut également omettre le `in` puisque la clôture n'est composée que de son corps.

# Clôtures

```
let tab = [5, 10, 54, 2]  
tab.sorted(by: >)  
//Retourne [54, 10, 5, 2]
```

Poussé à l'extrême. Swift définit l'opérateur `>` comme une fonction qui prend deux `Int` et retourne un `Bool`. Précisément le type attendu par notre clôture. On peut donc passer directement l'opérateur.

# Pour aller plus loin...

- <http://developer.apple.com/swift>
- [The Swift Programming Language \(iBook Store\)](#)
- [The Swift Programming Language \(Web\)](#)

