Swift

Jean-Baptiste Yunès

Jean-Baptiste.Yunes@gmail.fr

01/2020

Swift - les bases

Les variables, constantes, chaînes de caractères, tableaux et tuples

Les optionnels

Les opérateurs et structures de contrôle

Les clôtures et fonctions

Les classes, structures, énumérations et propriétés

L’héritage et les protocoles

Swift généralités

Le langage à employer pour développer sur plateformes Apple

Le développement de ce langage est une initiative d'Apple

Il remplace Objective-C (encore supporté) depuis 2014

Il est sous licence libre depuis 2015

Sa version courante est la 4.1 (2017)

Swift est multi-paradigme

programmation structurée, orientée objet et fonctionnelle

Sa syntaxe est plus commune

Objective-c était inspiré de Smalltalk (paradigme d'envoi de messages)

Swift utilise une vision plus répandue dans le monde des objets (appels de méthodes)

Il faut l'aborder de façon pragmatique

Il a été conçu de sorte à pouvoir être abordé par la pratique

Swift - l'inférence de type

Swift est un langage très fortement typé

toute variable est typée définitivement

aucune conversion entre type n'est effectuée implicitement

Swift possède un mécanisme d'inférence de type

permet de laisser le compilateur déterminer le type d'une variable

allège l'écriture

Swift - les identificateurs

Un identificateur valide est composé :

de lettres [a-zA-Z], de chiffres [0-9], du caractère *underscore* _
mais aussi de caractères Unicode du plan de base
grossièrement les caractères des jeux internationaux standard
de tous les caractères Unicode qui ne sont pas *privés*
ne peut pas commencer par un chiffre
il vaut mieux éviter d'utiliser des symboles trop exotiques
leur saisie n'est pas aisée

Exemples d'identificateurs valides :

monIdentificateur
user23
Δx
😡

Swift - les variables

La définition d'une variable

var x = 3

cette définition utilise l'inférence type

3 est un littéral du type Int

Le type Int est 32 bits ou 64 bits selon le type de la plateforme

La définition explicitement typée d'une variable

var x : Int

cette définition ne comporte pas d'initialisation

var x : Int = 3

c'est une définition typée avec initialisation

Une variable peut changer de valeur

x = 12+x

Swift - les types de nombres

Les types des nombres en Swift sont :

Sans taille déterminée (dépend de la plateforme)

Int

UInt

De taille spécifiée

Int8, Int16, Int32, Int64

UInt8, UInt16, UInt32, UInt64

Float, Double

Swift - les litéraux de nombres

Un littéral entier peut-être utilisé pour n'importe quel type entier

```
var i : Int = 34
var j : UInt16 = 34
```

Une littéral flottant peut-être utilisé pour n'importe quel type flottant

```
var f : Float = 3.4
var d : Double = 3.4
```

Si on utilise l'inférence de type :

```
var i = UInt(56)
var d = Double(45e3)
```

Swift - les littéraux de nombres

Pour aider à la lecture/écriture

On peut utiliser _ comme séparateur dans les littéraux de nombres

```
var unMilliard = 1_000_000_000
var mask16 = 0b_1111_0000_0101_1100
```

Swift - les littéraux de nombres

Pour les entiers

décimaux (sans préfixe)

123

binaires (préfixe 0b)

0b00110001

octaux (préfixe 0o)

0o734

hexadécimaux (préfixe 0x)

0x34AB

Pour les flottants

avec un point décimal ou un exposant

3

3.5

3e45

-3.4e-12

il existe d'autres littéraux plus *exotiques* de flottants (avec la base 16)

Swift - les bornes des types

Les types nombres entiers ont des valeurs minimales et maximales représentables

type.min

type.max

print(Int.min)

print(Int16.max)

Attention : l'arithmétique entière interdit les débordements!

var i = Int.min

i = i-1

provoque une erreur à l'exécution :

error: Execution was interrupted, reason: EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0).

On notera que les messages d'erreurs ne sont pas très explicites…

Swift - les constantes

Les constantes sont des variables ne pouvant être modifiées

Elles sont introduites par l'emploi du mot-clé let

```
let pi = 3.14
pi = 8.5 // interdit…
```

À part leur caractère constant qui empêche toute modification, elles s'emploient de la même manière

Swift - les chaînes de caractères

Swift possède le type String qui s'utilise assez naturellement

```
var hello = "Bonjour"
let bye : String = "Au revoir"
var salutation = String("Salut")
```

Il existe des littéraux multi-ligne

```
var texte = """
Bonjour,
Comment vas-tu ?
À bientôt
"""
```

il existe des variantes de formatage permettant d'intégrer correctement de tels littéraux dans le formatage du code

Swift - les chaînes de caractères

Tout caractère Unicode peut-être partie d'une chaîne de caractères

var chaîne = "abcd\"def\"\u{1BCA}ghi☠"

qui correspond à la chaîne

abcd"def"ᯊghi☠

La chaîne vide

var vide = ""

var vide = String()

Attention le type String est un type de valeur

Lorsqu'on passe un paramètre de type String on passe sa valeur… (voir fonctions plus loin)

Swift - les chaînes de caractères

La longueur d'une chaîne

```
var s = "Salut la compagnie"
print(s.count)
```

Swift - affichage console

Swift prédéfinit une fonction d'affichage :

print(*variable ou constante ou expression)*

Exemple :

```
var str = "Bonjour"
print(str)
```

Swift - le type caractère

var c : Character = "c"

Les littéraux de caractères sont des littéraux de chaînes réduites à un seul caractère!

Attention piège :

var s = "c"

s est une String ne contenant qu'un seul caractère

équivalent à

var s : String = "c"

Swift - les interpolations

La construction d'une chaîne paramétrée utilise l'interpolation

à l'intérieur d'une chaîne l'écriture \(*expression*) correspond à l'évaluation de l'*expression* et à sa substitution par sa valeur sous la forme d'une chaîne

Exemple :

```
var x = 12
var s = "La variable x vaut \(x)"
print(s)
print("La variable s vaut \"\(s)\"")
produit:
La variable x vaut 12
La variable s vaut "La variable x vaut 12"
```

Swift - l'indexation des chaînes

Swift utilise des mécanismes particuliers d'accès aux structures/ collections, etc

Le type String définit un sous-type String.Index permettant de désigner une position dans une chaîne

La manipulation de ce type peut surprendre…

L'indexation des caractères d'une chaîne s'effectue grâce à ce type

startIndex est une propriété des String permettant d'obtenir l'indice du début de chaîne

Exemple :

```
var s = "abcdef"
var idx = s.startIndex // idx est du type String.Index
print(s[idx])
```

produit :

a

Swift - l'indexation des chaînes

On peut obtenir un index à partir d'un index :

```
var s = "abcdef"
var first = s.startIndex
var second = s.index(after: first)
print(s[second])
var fourth = s.index(first, offsetBy: 3)
print(s[fourth])
```

produit :

```
b
d
```

Swift - l'indexation des chaînes

Les tentatives d'indexer une chaîne sur un caractère non existant provoque une erreur à l'exécution :

```
print(s[s.index(before:s.startIndex)])
```

produit :

```
Fatal error: cannot decrement before startIndex
```

endIndex représentent l'index de fin de chaîne, c'est-à-dire la position après le dernier caractère :

```
var idx = s.first
while idx != s.endIndex {
    print(s[idx])
    idx = s.index(after: idx)
}
```

Swift - l'indexation des chaînes

L'itération précédente peut-être obtenue avec la forme courte de l'itération généralisée des séquences :

```
for c in s {
    print(c)
}
```

Consulter la documentation de String pour en connaître toutes les subtilités…

Swift - les booléens

Le type booléen représente les valeurs de vérité, deux valeurs possible (qui ne sont pas des entiers!)

true

false

Les expressions de comparaison fournissent des valeurs booléennes

x == 3

x > 3

var b = x == 3

Les structures de contrôle nécessitent des booléens (voir plus loin)

Swift - les tableaux

Le type tableau est obtenu simplement par utilisation des []
Exemple :

```
var x = [1,2,3]
```

est un tableau d'Int, le type Swift est [Int]
On utilise l'indexation habituelle sur les tableaux :

```
print(x[2])
x[1] = 666
```

Attention, les tableaux définis *via* let sont constants, on ne peut les modifier!

```
let t = [1,2,3]
t[0] = 34
```

Erreur de compilation :
Cannot assign through subscript: 'aa' is a 'let' constant
L'accès à une case qui n'existe pas provoque une erreur à l'exécution :
Fatal error: Index out of range

Swift - les tableaux

Les tableaux non constants peuvent être modifiés (consulter la documentation pour en savoir plus)

```
var t = [1,2,3]
t.append(45)
t.insert(666, at: 2)
t.remove(at: 1)
t.removeAll()
```

Swift - les tableaux

Pour itérer sur les éléments d'un tableau on peut utiliser l'itération généralisée :

```
for e in t {
  print(e)
}
```

La création d'un tableau vide s'effectue avec ou sans l'inférence de type :

```
var t1 : [Int] = []
var t2 = [Int]()
```

On peut aussi créer un tableau d'une certaine taille en initialisation les éléments avec une valeur donnée :

```
var t = Array(repeating: 666, count: 150)
```

t est un tableau de 150 entiers initialisés à la valeur 666

Swift - les tuples

Swift permet la construction de types composés : les tuples

(3,"Bad")

est un tuple (ici un couple)

On peut définir une variable dont le type est un tuple :

```
var t = (3,"Bad")
var t2 = ("Hello",34,4.5,"Cool")
```

Le type de t est (Int,String), celui de t2 est (String,Int,Double,String)

Swift - les tuples

L'extraction d'une des valeurs d'un tuple peut s'effectuer à l'aide d'indices entiers :

```
var t = ("a","b",3,"blabla")
print(t.1)
```

produit :

```
b
```

Si l'on désire on peut aussi nommer les éléments du tuple :

```
var t = (name: "Smith", age: 45)
print("\(t.name) est âgé de \(t.1) ans")
```

C'est très utile pour renvoyer plusieurs valeurs en retour d'une fonction…

Swift - les optionnels

Les variables et constantes telles que définies précédemment désignent nécessairement une valeur ou un objet.

Elle ne sont pas conçues pour pour désigner l'absence (i.e.: «pointeur» null par exemple ou «pas de valeur» avec une quelconque convention…)

Swift propose pour cela les types optionnels (optionals)

Un type optionnel est obtenue par dérivation d'un type :

Int?

représente le type entier optionnel

Un Int? désigne soit un entier valide soit pas d'entier

L'absence de valeur est représentée par la constante nil

Swift - les optionnels

```
var x : Int? = nil
var y : Int? = 3
```

Dans les deux cas, le type est entier optionnel, pas entier…

Ce type est employé partout où il est possible qu'un entier ne puisse pas être construit/représenté…

Par exemple :

```
var text = "123"
var x = Int(text)
print(x)
text = "blabla"
x = Int(text)
print(x)
produit :
Optional(123)
nil
```

La conversion peut ne pas aboutir, la valeur de retour est donc soit la conversion est impossible et c'est nil, soit c'est possible et c'est l'optionnel qui encapsule la valeur.

Swift - les optionnels

On peut donc tester si l'optionnel désigne une valeur ou non

Par exemple :

```
var x : Int? = ...
if x != nil {
  // il y a bien une valeur
} else {
  // il n'y a pas de valeur
}
```

Swift - les optionnels

S'il y a une valeur on peut la sortir de l'encapsulation en utilisant le déréférençement forcé :

optionnel!

permet d'obtenir la valeur du type non optionnel si elle existe. Attention, si l'on tente de faire ressortir une valeur depuis l'optionnel nil, une erreur à l'exécution est produite…

Par exemple :

```
if x != nil {
  print("La valeur est \(x!)")
} else {
  // il n'y a pas de valeur
}
```

Cette forme n'est pas ce qui est recommandé…

Swift - les optionnels

On utilise plutôt la liaison optionnelle.

Par exemple :

```
var x : Int?
if let v = x {
  print("La valeur est \(v)")
} else {
}
```

La forme if let permet d'obtenir une constante contenant la valeur contenue dans l'optionnel (i.e. la valeur «sort» de l'optionnel et est capturée dans la constante)

Swift - les optionnels

Dans certains cas, on sait que passé un certain point du programme un optionnel contiendra toujours une valeur (la logique du programme le dit)

Pour éviter d'alourdir son utilisation, on peut utiliser un optionnel implicitement désencapsulé (implicit unwrapped optional).

```
var x : Int! = nil
print(x) // potentiellement dangereux
x = x+3 // erreur exécution (nil…)
x = 12
print(x) // sans danger
x = x+3 // sans danger
```

L'erreur produite est :

```
Fatal error: Unexpectedly found nil while unwrapping an Optional value
```

Swift - les opérateurs

Les opérateurs de base sont «standard»

Affectation

=

Arithmétique

+ - * / % (unaire + et -)

Composés

+= -= *= /= %=

Comparaisons

> >= < <= == !=

Logiques

! && || (ils sont fainéants - lazy)

Note : les comparaisons sont étendues naturellement aux tuples

Swift - les opérateurs

```
let x = (2,"bonjour")
let y = (3,"au revoir")
let z = (2, "au revoir")
if x < y {
  print(x)
} else {
  print(y)
}
if x < z {
  print(x)
} else {
  print(z)
}
produit :
(2, "bonjour")
(2, "au revoir")
```

Swift - les opérateurs

L'opérateur ternaire

permet de remplacer les formes comme :

```
if test {
  width = 23
} else {
  width = 45
}
```

par

```
width = test ? 23 : 45
```

Swift - les opérateurs

L'opérateur ternaire est parfois employé avec les optionnels :

```
a != nil ? a! : b
```

Si pas de valeur, valeur par défaut b sinon la valeur de a

Il existe un opérateur Swift permettant de l'écrire explicitement :

```
a ?? b
```

Swift - les structures de contrôle

La boucle for traditionnelle est interdite depuis Swift 3, il ne reste plus que l'itération généralisée dite for-in

Elle permet d'itérer sur de nombreuses constructions.

Itération sur une collection :

```
for e in collection {
  print(e)
}
```

permet de parcourir tous les éléments de la collection et d'effectuer le corps de la boucle pour chacun d'entre eux capturés par la variable e

Note : e est du type des éléments de la collection…

Swift - les structures de contrôle

On peut utiliser les for-in pour itérer sur des intervalles numériques (numeric ranges).

Un intervalle peut être défini par:

1...10

ce qui signifie tous les nombres entiers de 1 à 10 (y compris 10)

ou par :

1..<10

ce qui signifie tous les nombres entiers de 1 à 10 (10 non compris)

La boucle prend alors la forme, par exemple :

```
for i in 1...10 {
  print(i)
}
```

Swift - les structures de contrôle

Attention les intervalles doivent être corrects, c'est-à-dire croissants et non vides…

```
for i in 5...2 {
  print(i)
}
```

est interdit! Cela provoque une erreur à l'exécution :

```
Fatal error: Can't form Range with upperBound < lowerBound
```

On peut contrôler le pas en utilisant un «stride»:

```
for i in stride(from: 1, to: 23, by: 5) {
  print(i)
}
```

il existe aussi des intervalles plus exotiques, les ouverts…

Swift - les structures de contrôle

Si l'on utilise pas la variable de contrôle dans la boucle, comme dans :

```
for i in 1...10 {
  print("bonjour")
}
```

il est fortement conseillé d'utiliser la variable anonyme spéciale :

```
for _ in 1...10 {
  print("bonjour")
}
```

Cette variable est inemployable ailleurs qu'à sa définition.

Swift - les structures de contrôle

Les intervalles (possiblement ouverts) peuvent aussi être utilisés naturellement avec les tableaux :

```
var t = Array(repeating: 666, count: 2000)
for e in t[...500] {
    print(e)
}
for e in t[200...213] {...}
for e in t[450...] {...}
```

Les intervalles sont des types, on peut donc déclarer des intervalles :

```
let i = 300...400
for j in i {...}
```

Swift - les structures de contrôle

La boucle while s’écrit :

```
while expression-booléenne {
    corps
}
```

La boucle repeat-while s’écrit :

```
repeat {
    corps
} while expression-booléenne
```

Swift - les structures de contrôle

L'alternative s'écrit :

if *expression-booléenne* {
    corps-si-vrai
} else {
    corps-si-faux
}

La partie else est évidemment optionnelle…

On notera que les expressions simples n'ont pas besoin d'être parenthésées…i.e. on écrit :

```
if x == 3 {…}
```

plutôt que :

```
if (x == 3) {…}
```

Même remarque pour les autres boucles…

Swift - les structures de contrôle

Le branchement multiple (l'aiguillage) switch.

switch *expression* {
case *valeur1*:
  corps1
case *valeur2*:
  corps2
...
default:
}

Bien que le default puisse ne pas être employé, il est très souvent nécessaire :

Attention, le switch doit assurer que toutes les valeurs possibles sont traitées! (d'où l'emploi souvent nécessaire de default)

Note : break n'est pas utile, chaque cas est par défaut indépendant des autres…

Swift - structures de contrôle

Exemple:

```
let c : Character = "r"
switch c {
case "a":
    print("Ok c'est a")
case "b":
    print("Bien c'est b")
case "z":
    print("dernière de l'alphabet")
default:
    print("tous les autres cas")
}
```

Si on ne met pas le default, le compilateur émet l’erreur :

Switch must be exhaustive

Swift - les structures de contrôle

Les cas ne peuvent être vides!

Si on désire vraiment un cas vide, il faut y placer break

Si on désire que deux cas (ou plus) utilisent le même bloc il faut composer les cas.

Ainsi :

```
switch c {
case "a": // cas vide
    break
case "b", "B", "c": // cas composé
    corps
default:
    corps
}
```

Swift - les structures de contrôle

On peut même y utiliser des intervalles…

```
var x = 127

switch x {
case 0:
    print("nul!")
case 1..<10:
    print("unités")
case 10..<100:
    print("dizaines")
case 100..<1000:
    print("centaines")
case 1000..<10_000:
    print("milliers")
default:
    print("beaucoup...")
}
```

Swift - les structures de contrôle

On peut même utiliser des intervalles et des tuples et capturer les valeurs que l'on souhaite

```
var t = (400,"message")
switch t {
case (0, let m):
    print("ok \(m)")
case (1..<6, let m):
    print("error \(m)")
case (6..., let m):
    print("warning \(m)")
default:
    break
}
```

Il existe des constructions encore plus complexes…

Swift - les structures de contrôle

Les transferts du point de contrôle sont :

break, continue, fallthrough, return et throw

Break provoque la sortie immédiate de la structure en cours, boucle ou switch

Continue provoque la reprise immédiate de la boucle en cours

Fallthrough provoque la continuation vers le cas suivant dans un switch

Return et throw sont utilisés respectivement pour retourner depuis une fonction vers l'appellant et throw pour lever une exception

Note : break et continue peuvent utiliser une étiquette afin de désigner la boucle à briser ou reprendre

Swift - les fonctions

Comme dans tous les langages les fonctions prennent des paramètres et renvoient des valeurs (possiblement).

En Swift elles doivent être typées (type des arguments et type de retour).

On utilise le mot-clé func pour définir une fonction.

Exemple :

```
func addition(a: Int, b: Int) -> Int {
    return a+b
}
```

Le type de cette fonction est :

```
(Int,Int)->Int
```

Les fonctions sont des objets comme les autres! On peut les manipuler en tant que telles (le langage est fonctionnel).

On dit que ce sont des valeurs de première classe.

Swift - les fonctions

Attention, si l'on définit :

```
func addition(a: Int, b: Int) -> Int {
    return a+b
}
```

Cela impose de nommer les arguments à l'appel! Comme :

```
print("Le résultat est \(addition(a:3, b:5))")
```

Ne pas se méprendre, cela ne permet pas de «mélanger» les arguments, il faut respecter l'ordre…

Swift - les fonctions

Une fonction sans paramètre :

```
func f() -> Int {
    return 34
}
```

Attention, une fonction sans valeur de retour :

```
func g() {
    return
}
```

Pas de -> ni de spécification de type de retour…

Swift - les fonctions

Si l'on souhaite renvoyer plusieurs valeurs, on peut renvoyer un tuple :

```
func addAndSub(a: Int, b: Int) -> (sum: Int, sub: Int) {
  return (a+b, a-b)
}
print(addAndSub(a: 10, b: 3))
```

produit:

(sum: 13, sub: 7)

Swift - les fonctions

Les paramètres peuvent avoir deux noms, un nom externe utilisé à l'appel et qui permet de «documenter» l'usage de ce paramètre et un usage «interne» pour manipuler la valeur passée à l'appel :

```
func divide(dividende n: Int, diviseur d: Int) -> Double {
    return Double(n)/Double(d)
}
print(divide(dividende: 4, diviseur: 7))
func distanceAuPointZéro(abscisse x: Double, ordonnée y:
Double) -> Double {
    return (x*x+y*y).squareRoot()
}
print(distanceAuPointZéro(abscisse: 3.4, ordonnée: 7.8))
```

Swift - les fonctions

Si l'on souhaite omettre le nom externe (dans le cas où le rôle de l'argument est évident) on peut utiliser l'anonymisation :

```
func add(_ x: Int,_ y: Int) -> Int {
    return x+y
}
print(add(4,5))
```

Swift - les fonctions

On peut déclarer un argument variadique (variadic parameter), c'est-à-dire un argument pouvant recevoir plusieurs valeurs du type (au moins une). Cet argument pouvant être placé n'importe où dans la liste :

```
func sum(elements: Int...) -> Int {
    var s = 0
    for e in elements { s += e }
    return s
}
print(sum(elements: 1, 2, 3, 4))
print(sum(elements: 2))
print(sum(elements: 1, 20, 3, 4, 70))
```

Swift - les fonctions

Par défaut les arguments reçus sont des constantes… Une fonction ne peut donc pas modifier ses arguments (c'est donc bien une fonction et non une procédure). Si on désire le contraire, on peut l'indiquer, cela change la définition de la fonction et son appel :

```
func modifie(_ i: inout Int) {
    i = 666
}
var x = 0;
modifie(&x) // x est passé en inout il faut donc écrire &x
print(x)
```

Swift - les fonctions

Les fonctions sont des objets comme les autres, on peut déclarer des variables de type fonction, etc.

```
func inc(_ i:Int) -> Int { return i+1 }
func dec(_ i:Int) -> Int { return i-1 }
var f : (Int)->Int
f = inc
print(f(5))
f = dec
print(f(5))
```

Swift - les fonctions

On peut les passer en paramètre, les renvoyer en retour, etc.

```
func inc(_ i:Int) -> Int { return i+1 }
func dec(_ i:Int) -> Int { return i-1 }
func compose(f: (Int)->Int, g: (Int)->Int, x: Int) -> Int {
    return f(g(x))
}
print(compose(f:inc, g:inc, x:3)) // 5
print(compose(f:inc, g:dec, x:3)) // 3
print(compose(f:dec, g:dec, x:3)) // 1
```

Swift - les fonctions

On peut spécifier une valeur par défaut aux paramètres :

```
func f(a: Int, b: Int=23) { print("\(a) \(b)") }
func g(a: Int = 33, b: Int) { print("\(a) \(b)") }
func h(a: Int,  b: Int = 33, c: Int) { print("\(a) \(b) \(c)") }
f(a:22)
f(a:22, b:11)
g(b: 99)
g(a: 0, b: 99)
h(a:1, c:9)
```

Attention si les paramètres n’ont pas de nom externe

Swift - les clôtures

Les clôtures Swift (closures) sont des blocs de code anonymes (Objective-C blocks ou lambdas C++/Haskell) pouvant capturer des objets de leur environnement.

La capture de l'environnement fonctionne déjà avec les fonctions ordinaires (qui sont des clôtures nommées)…

```
var v = 10
func f(_ i: Int) -> Int {
    return i+v // v est capturée par la clôture
}
print(f(5))
v = 20
print(f(5))
```

Swift - les clôtures

Si l'on a une fonction qui prend en paramètre une fonction :

```
func f(_ i: Int, combine: (Int)->Int) -> Int {
    return combine(i)
}
var v = 10
func c(_ i: Int) -> Int {
    return i+v
}
print(f(5,combine: c))
v = 20
print(f(5,combine: c)
```

Swift - les clôtures

Si la fonction n'est utilisée qu'une seule fois on peut alors définir une clôture qui la représente :

```
print(f(5, combine: { (x:Int) -> Int in return x+v }))
```

La valeur :

```
{ (x:Int) -> Int in return x+v }
```

est une clôture, c'est une fonction (sans nom) qui prend un entier, renvoie un entier et calcule la somme de son argument avec la valeur capturée

Swift - les clôtures

Ainsi la définition d'une fonction est équivalente à la définition d'une variable dont la valeur est une clôture…

```
func f(_ a:Int) -> Int { return a+1 }
```

est équivalent à :

```
let f = { (a: Int) -> Int in return a+1 }
```

Swift - les clôtures

Si la clôture est définie dans un contexte suffisamment riche on peut alors utiliser l'inférence de type pour simplifier sa définition

```
print(f(5, combine: { (x:Int) -> Int in return x+v }))
```

Le contexte indique que l'argument combine attendu est (Int)-> Int on peut donc enlever ces informations :

```
print(f(5, combine: { (x) in return x+v }))
```

Le calcul est réduit à une simple expression on peut donc réduire l'expression à :

```
print(f(5, combine: { (x) in x+v }))
```

Il n'y a qu'un seul argument on peut donc alléger encore :

```
print(f(5, combine: { x in x+v }))
```

Swift - les clôtures

Il existe par ailleurs une façon de nommer les arguments attendus si on ne désire pas leur attribuer un nom particulier, chacun s'appelle dans l'ordre $0, $1, etc comme ici il n'y en a qu'un on obtient :

```
print(f(5, combine: { $0+v }))
```

D'autre part, puisque la fonction est attendue comme dernier paramètre on peut écrire :

```
print( f(5) { $0+v } )
```

Il existe d'autres constructions subtiles avec les clôtures, l'échappement (escaping) et la clôture automatisée (autoclosure). Ce sont des concepts avancés, on laissera le lecteur les découvrir si nécessaire…

Swift - les classes

Les classes (avec les structures) sont les blocs de construction les plus utilisés dans la programmation objet.

Une classe Swift définit essentiellement :

des propriétés (données)
des méthodes (opérations)
des initialiseurs

La définition d'une classe a la forme :

class *identificateur* {
    définition du contenant
}

Swift - les classes

Une définition comme :

```
class Maison {
    var surface = 100
    var etages = 1
    var petitNom : String?
}
```

autorise l'instanciation suivante :

```
let m = Maison()
```

La définition précédente correspond à la définition d'objets ayant comme propriétés une surface, un nombre d'étages et un surnom. Ces propriétés ayant des valeurs par défaut, respectivement 100, 1 et nil (le surnom est optionnel…).

Swift - les classes

L'accès à une propriété s'effectue en utilisant la notation pointée :

```
print("""
    La maison \(m.petitNom ?? "sans nom") \
    a \(m.etages) étage(s) et \
    une surface de \(m.surface) m2
    """)
```

Si l'on a accès à une propriété variable on peut la modifier :

```
// agrandissement
m.surface = 120
```

Les propriétés sont propres aux instances.

Swift - les classes

Les instances de classes sont manipulées par des types références. L'affectation ne copie pas les objets…

```
var m2 = m
print(m.surface)
print(m2.surface)
m2.surface = 140
print(m.surface)
print(m2.surface)
```

fournit le même résultat. m2 et m désignent toutes deux le même objet.

Conséquence : les objets sont transmis par référence.

Swift - les classes

Les constantes de références sont simplement des références qui désignent toujours le même objet (cela ne signifie pas que l'on ne puisse pas modifier l'objet référencé…)

```
let m3 = m
m3.surface = 180
m3 = Maison() // ceci est interdit…
```

Swift - les classes

Les tests sur les références sont particuliers.

Il existe notamment deux opérateurs :

==

qui teste si les objets référencés sont «équivalents», c'est-à-dire si leur contenu peut être considéré comme identique (cette notion est laissée libre d'appréciation aux concepteurs de la classe). Par défaut cet opérateur n'existe pas…

et

===

qui teste si les deux références référencent le même objet

Swift - les classes

```
let m1 = Maison()
let m2 = m1
if m1 === m2 { print("une seule maison") }
else { print("deux maisons") }
let m3 = Maison()
if m1 === m3 { print("une seule maison") }
else { print("deux maisons") }
```

produit :

```
une seule maison
deux maisons
```

Swift - les classes

Les éléments de base constituant les classes sont les propriétés stockées

les propriétés stockées peuvent être variables ou constantes

Par exemple :

```
class Maison {
   var surface = 100
   let annéeDeConstruction = 2000
}
```

comme pour les constantes cela signifie que la date de construction ne peut être modifiée.

Swift - les classes

Les propriétés stockées peuvent être lazy, c'est-à-dire que leur contenu n'est initialisé que lorsqu'on les utilisent pour la première fois. Par conséquent, si on ne les utilisent pas, elles n'occupent pas l'espace qui leur serait par ailleurs nécessaire…

```
func load() -> [UInt8] {
    print("Loading data")
    return [0xFF,0xFF] // whatever
}
class Image {
    lazy var data = load()
    let name = "i1"
}
let image = Image()
print(image)
print(image.name)
print(image.data)
```

produit :

```
__lldb_expr_179.Image
i1
Loading data
[255, 255]
```

Swift - les classes

Il existe d'autres propriétés : les propriétés calculées

L'affectation ou la lecture sont représentées par du code

```
class K {
  var p : Int { // readonly property
    get { return 3 }
  }
}
let k = K()
print(k.p)
```

ou en forme courte

```
class K {
  var p : Int { return 3 }
  }
}
```

Swift - les classes

Il existe d'autres propriétés : les propriétés calculées

L'affectation ou la lecture sont représentées par du code

```
class K {
  var p : Int {
    get { return 3 }
    set (value) { print("Setting to \(value)") }
  }
}
let k = K()
print(k.p)
k.p = 666
```

Swift - les classes

Il existe aussi des observateurs de propriétés stockées

Ce sont des fonctions qui permettent de capturer les écritures de propriétés stockées

```
class K {
  var p : Int = 0 { // Warning : propriété stockée!
    willSet (nValue) { print("avant vaut \(p) va être changé
en \(nValue)" }
    didSet { print("après vaut \(p) mais valait \(oldValue)") }
  }
}
let k = K()
k.p = 666
```

Swift - les classes

L'autre élément de base de construction des classes est la méthode. On parle ici de méthode d'instance, c'est-à-dire les méthodes qui agissent sur les objets.

La notation pointée est utilisée et la syntaxe est exactement celle d'appel de fonction. Le contexte est l'objet sur lequel la méthode est appelée et agit.

```
class Maison {
    var surface = 100
    func agrandir(surfaceAdditionnelle s: Int) {
        surface += s
    }
}
let m1 = Maison()
print(m1.surface)
m1.agrandir(surfaceAdditionnelle: 10)
print(m1.surface)
```

produit :

```
100
110
```

Swift - les classes

Chaque instance possède une propriété implicitement définie et qui le désigne lui-même : self (dans la plupart des autres langage, le mot-clé est this).

```
class Maison {
    var surface = 100
    func agrandir(surfaceAdditionnelle surface: Int) {
        self.surface += surface
    }
}
```

Attention : contrairement à Objective-C, il est impossible de modifier self dans les classes (pour les structures c'est une autre histoire…). Toute tentative conduit au message de compilation :

Cannot assign to value: 'self' is immutable

Swift - les classes

Si l'on souhaite initialiser les propriétés avec des valeurs fournies à l'instanciation il faut utiliser des initialiseurs. Swift utilise le terme initializer et non constructor. Il s'agit bien d'initialiser… Si des propriétés ne sont pas initialisées par défaut, Swift requiert des initialiseurs.
Par exemple :

```
class Maison {
    var surface : Int
}
```

produit l'erreur de compilation :

Class 'Maison' has no initializers

Swift - les classes

Les initialiseurs sont spécifiés *via* le mot-clé init, attention ce ne sont pas des fonctions, on utilise pas le mot-clé func :

```
class Maison {
    var surface : Int
    init(surface s : Int) {
        surface = s
    }
}
```

L'instanciation prend alors la forme :

```
let homeSweetHome = Maison(surface: 100)
```

Swift - les classes

Swift s'assure que les propriétés sont toujours initialisées (par défaut ou *via* un initialiseur) :

```
class Maison {
    var surface : Int
    var etages = 1
    init(surface s:Int) { surface = s }
    init(surface s:Int, etages e:Int) { surface = s; etages = e }
}
let hsh = Maison(surface: 100)
let vacances = Maison(surface: 120, etages: 3)
```

Les types optionnels sont initialisés par défaut à nil.

Attention, les constantes sont nécessairement initialisées une et une seule fois (par défaut ou *via* init)

Attention, dès qu'un initialiseur est défini, l'initialiseur par défaut n'est plus disponible.

Swift - les classes

Les initialiseurs tels que définis précédemment sont appelés initialiseurs désignés (designated initializers).

Lors de l'instanciation, un et un seul initialiseur désigné peut être appelé.

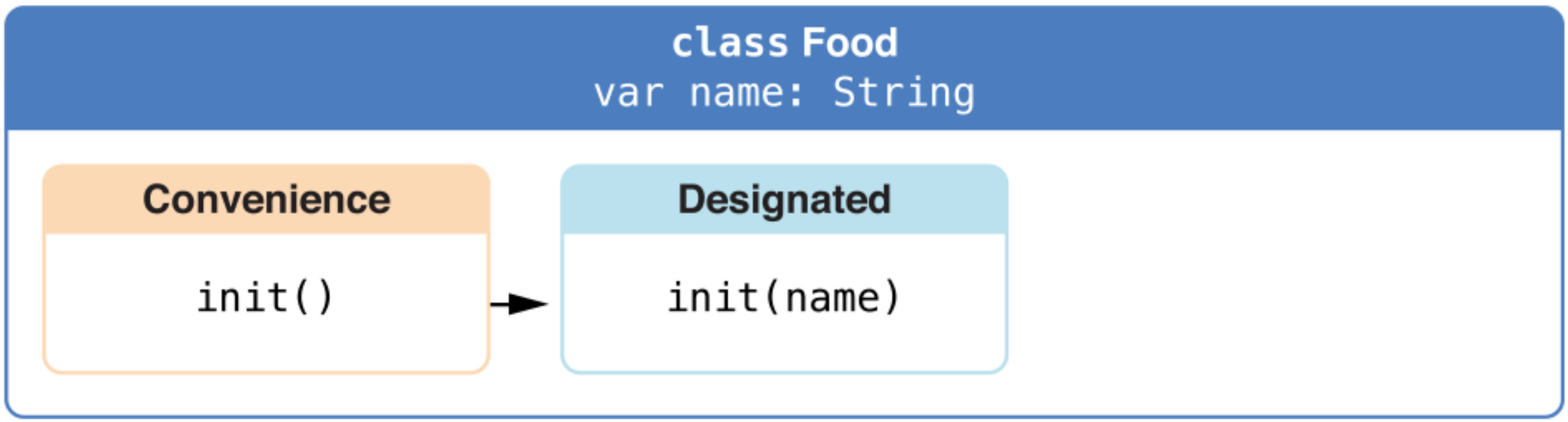
Si l'on souhaite fournir des façons d'initialiser pratiques (afin de factoriser les initialisations), il faut fournir des initialiseurs pratiques (convenience initializers).

Un convenience initializer peut appeler un autre convenience ou un designated (*via* self.init).

Un designated ne peut appeler un autre quelqu'il soit (sauf si héritage).

Swift - les classes

Le schéma suivant illustre le mécanisme :



extrait de la documentation Apple

Swift - les classes

On peut définir des initialiseurs qui échouent (failable initializers). Ils renvoient donc un optionnel du type, la syntaxe est init?. Si l'initialisation échoue ils renvoient nil, sinon ils initialisent les propriétés :

```
class Maison {
    var surface : UInt
    init?(surface s: UInt) {
        if (s>1_000) { return nil }
        surface = s
    }
}
let m1 = Maison(surface: 10_000)
let m2 = Maison(surface: 25)
```

m1 et m2 sont tous les deux du type Maison?

Swift - les classes

```
class Maison {
    var surface : Int
    var etages : Int
    init(surface s: Int, etages e: Int) {
        surface = s
        etages = e
    }
    convenience init(surface s: Int) {
        self.init(surface: s, etages: 1)
    }
    convenience init() {
        self.init(surface: 100)
}
```

Swift - les structures

Les structures sont des types valeurs, on les manipulent directement et non pas *via* des références.

Par conséquent elles sont copiées *via* les affectations ou les passages de paramètres.

Les structures peuvent comme les classes contenir des propriétés et des méthodes.

Des initialiseurs sont automatiquement synthétisés si aucun n’est défini et qui permettent d’initiliser toutes les propriétés (les memberwise initializers)

Attention, les initialiseurs définis font disparaître les initialiseurs synthétisés.

Swift - les structures

```
struct Point {
    var x : Int = 0
    var y : Int = 0
}

var p = Point(x: 10, y: 20)
print("\(p.x) \(p.y)")

p = Point()
print("\(p.x) \(p.y)")

var p2 = p
p.x = 666
print("\(p.x) \(p.y)")
print("\(p2.x) \(p2.y)")
```

produit :

```
10 20
0 0
666 0
0 0
```

Swift - les structures

Attention, dans une méthode de structure self est considéré comme non mutable par défaut, i.e. on ne peut modifier les propriétés.

Si on souhaite modifier les propriétés il faut définir la méthode comme mutante (mutating) :

```
struct Point {
    var x = 10
    var y = 10
    mutating func moveRight() {
        x += 10
    }
}
```

Swift - les structures

On peut modifier self dans les mutantes!

```
class Strange {
    mutating func rebuild() {
        self = Strange()
    }
}
```

Swift - les énumérations

Un type énumération permet de définir un ensemble de valeurs.

```
enum JourDeLaSemaine {
    case lundi
    case mardi
    case mercredi
    case jeudi
    case vendredi
    case samedi
    case dimanche
}
let l = JourDeLaSemaine.lundi
```

Si le contexte est suffisamment riche, l'inférence de type permet de raccourcir l'écriture :

```
let j : JourDeLaSemaine = .jeudi
```

Swift - les éunmérations

On peut aussi définir l'énumération précédente par :

```
enum JDLS {
    case lundi, mardi, mercredi, jeudi, vendredi, samedi,
dimanche
}
```

L'affichage produit la chaîne qui représente le littéral :

```
let l = JDLS.lundi
print(l)
```

affiche :

```
lundi
```

Swift - les énumérations

Le contrôle switch peut utiliser les énumérations :

```
var j : JDLS = …
switch j {
case .lundi:
    print("monday")
default:
    print("not monday")
}
```

Swift - les énumérations

Les valeurs des énumérations peuvent contenir des valeurs associées (associated values) :

```
enum ID {
    case local(UInt32)
    case international(countryCode: UInt16,UInt32)
}

let l = ID.local(0x0F0F0F0F)
let i = ID.international(countryCode: 257, 0x77777777)

func showID(_ id: ID) {
    switch id {
    case .local(let id):
        print("local id \(id)")
    case .international(let countryCode, let id):
        print("id \(id) for country \(countryCode)")
    }
}

showID(i)
showID(l)
```

Swift - les énumérations

On peut aussi utiliser des valeurs brutes (raw values). Cela permet d'obtenir des valeurs d'énumération pour représenter des valeurs particulières d'un type sous-jacent :

```
enum Constants : Double {
    case e = 2.718
    case pi = 3.1415
    case phi = 1.618
}

let myConstant = Constants.e
print(myConstant)
print(myConstant.rawValue)
if let mc = Constants.init(rawValue: 3.145) {
    print(mc)
} else {
    print("unknown constant")
}
```

Swift - les protocoles

Le concept Swift de protocole (protocol) correspond à la notion d'interface en Java ou classe purement abstraite en C++.

Un protocole définit un contrat d'implémentation de méthode et de propriétés (plus surprenant).

```
protocol Printable {
    func print()
}
```

Swift - les protocoles

Une classe qui désire se conformer à un protocole le fait en précisant la liste des protocoles à qui se conformer ainsi que l'implémentation des méthodes :

```
class A : Printable {
    func print() {
        Swift.print("A") // Swift. pour aider Swift à comprendre
    }
}
```

La relation d'implémentation est une relation de sous-typage : A est sous-type de Printable.

D'autre part, Printable est un type valide pour une variable :

```
let p : Printable = A()
p.print()
```

Swift - les protocoles

Un protocole peut être sous-type d'autres protocoles

```
protocol P : Printable, Stackable, Runnable, Doable {
    func f(a: Int) -> Int
}
```

Un protocole peut imposer une méthode mutante.

Les structures peuvent se conformer à un protocole :

```
protocol Modifiable {
    mutating change()
}
struct S : Modifiable {
    mutating change() {
        ...
    }
}
```

Swift - les protocoles

Un protocole peut imposer l'existence d'une propriété et d'imposer les opérations possibles (get et/ou set) :

```
protocol Describable {
    var asString : String { get } // { get set }
}
class K : Describable {
    var asString = "I am what I am"
}
let d : Describable = K()
print(d.asString)
```

Swift - l'héritage

Une classe peut étendre au plus une autre classe (pas d'héritage multiple en Swift).

```
class K {
}
class SK : K {
}
```

On peut aussi implémenter des protocoles :

```
class SK2 : K, Printable, Hashable {
}
```

Swift - l'héritage

Swift ne fournit pas d'héritage par défaut. Il n'y a donc pas de classe de base commune.

Toute classe définie sans héritage est dite classe de base (base class)

Toutefois lors de développement dans l'écosystème Apple, il est recommandé d'hériter de NSObject, cela évite les problèmes futurs d'interaction des classes avec les frameworks. Cela nécessite l'importation du framework correspondant :

```
import Foundation // classes de base de l'écosystème
```

Swift - l'héritage

La redéfinition de tout constituant des classes héritées est possible mais nécessite l'emploi du mot-clé override :

```
class K {
    func f() { }
}
class SK : K {
    override func f() { }
}
```

Attention, les propriétés stockées ne peuvent être redéfinies que par des propriétés calculées (voir plus loin)

Swift - l'héritage

L'utilisation d'un constituant hérité nécessite l'emploi du mot-clé super

```
class K {
    func f() { }
}
class SK : K {
    override func f() { super.f() }
}
```

Swift - l'héritage

Pour empêcher la redéfinition d'un constituant par une sous-classe il faut le qualifier de final

```
class K {
    final func f() { }
}
class SK : K {
    func g() { }
}
```

Swift - l'héritage

L'héritage et initialisation.

```
class K {
    var p : Int
    init(_ v: Int) { p = v }
}
class SK : K {
    var pp : Int
    init(_ v : Int, _ vv : Int) {
        pp = vv
        super.init(v) // appel de l'initialiseur de la super classe
    }
}
```

Attention, avant d'appeler l'initialiseur de la super-classe il est impératif d'avoir initialisé toutes les propriétés de la classe! C'est l'inverse de beaucoup de langages orientés objets.

Swift - l'héritage

Les initialiseurs de la super classe sont hérités si la sous-classe ne définit pas d'initialiseur :

```
class K {
    var pk : Int
    init(_ v=: Int) { pk = v }
}
class SK : K {
    var psk : Int = 20
}
let sk = SK(12)
```

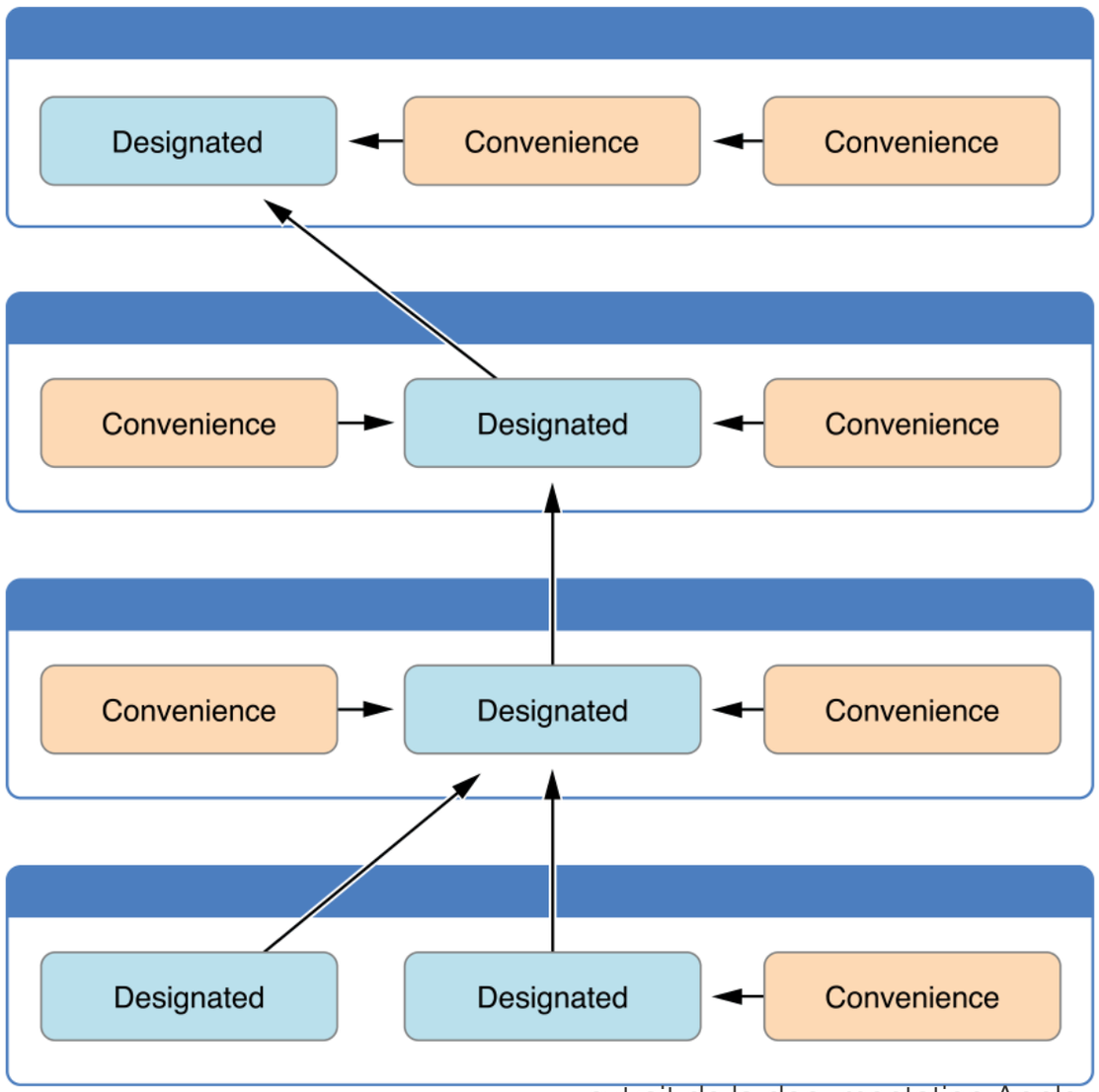
Swift - l'héritage

Une classe peut imposer qu'une sous-classe implémente un (ou plusieurs) initialiseur(s) particulier (required init) dans le cas ou elle redéfinit au moins un initialiseur :

```
func loadFromFile() -> Int { return 20 }
class K {
    var pk : Int
    init(_ v: Int) { pk = v }
    required init(fromFile: String) { pk = loadFromFile() }
}
class SK : K {
    var psk : Int = 20
    init() { super.init(10) }
    required init(fromFile: String) {
        psk = loadFromFile()
        super.init(fromFile: "toto.txt")
    }
}
let sk = SK()
```

Swift - l'héritage

Les règles d'appel des initialiseurs dans le cas de l'héritage sont résumées par :



extrait de la documentation Apple

Swift - l'héritage

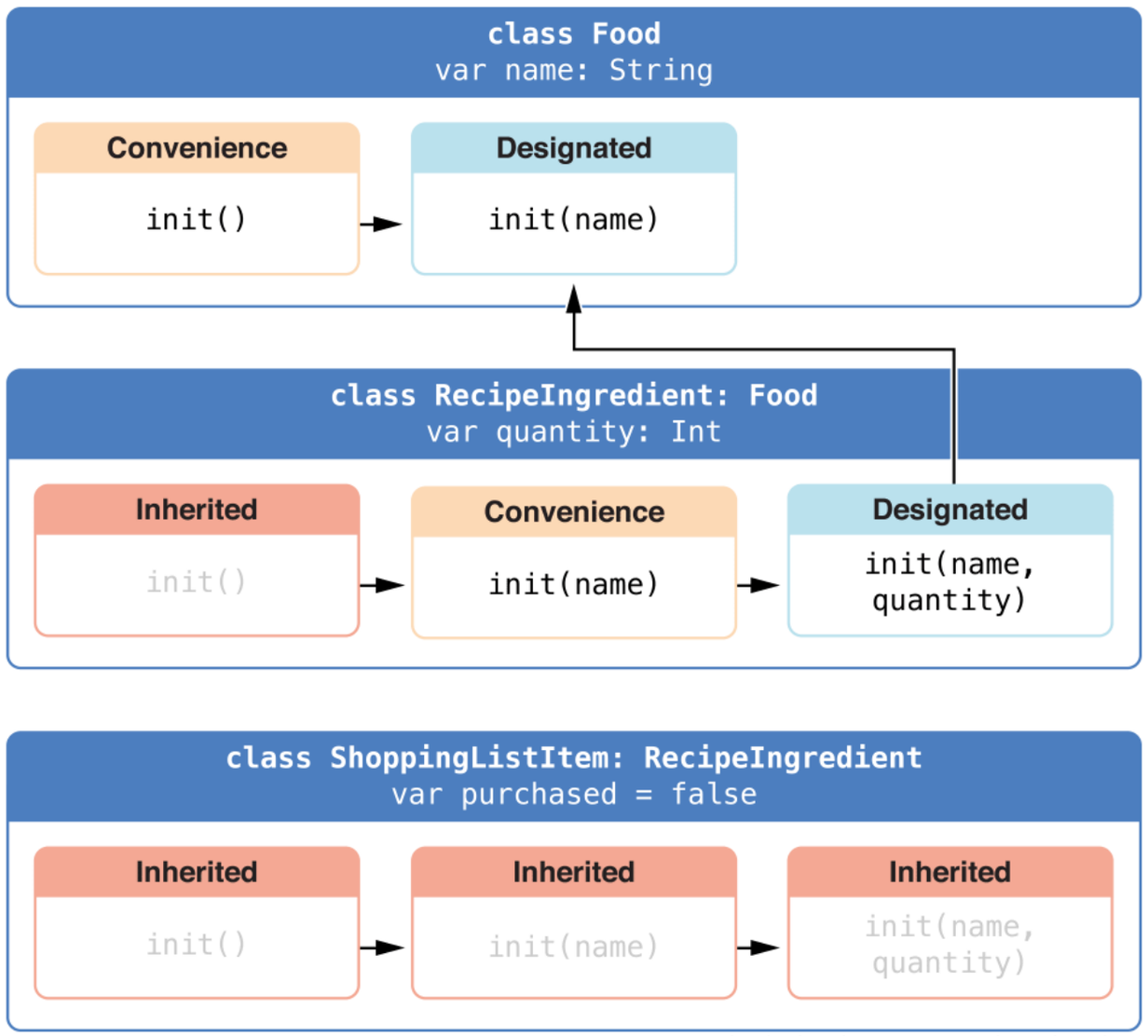
L'héritage est assez complexe :

```
func loadFromFile() -> Int { return 20 }
class K {
    var pk : Int
    convenience init() { self.init(20) }
    init(_ v: Int) { pk = v }
}
class SK : K {
    var psk : Int = 20
    override init(_ v: Int) {
        psk = 666; super.init(999)
    }
}
let sk = SK()
```

La règle à l'œuvre ici est si une sous-classe redéfinit tous les initialiseurs désignés (designated) de sa super classe, elle hérite de tous ses initialiseurs pratiques (convenience)

Swift - l'héritage

Une chaîne compliquée serait :



extrait de la documentation Apple

Swift - déinitialisation

Swift gère automatiquement la mémoire. Il y a un garbage collector particulier : un compteur de référence. Celui-ci nécessite quelques précautions, mais pour les cas simple il n'y a pas à s'en préoccuper.

Lorsqu'un objet est recyclé il peut-être nécessaire de libérer des ressources détenus par l'objet.

Si elle existe, il y a appel de la déinitialisation, un bloc de code marqué par le mot-clé deinit ce n'est pas une fonction…

Swift - déinitialisation

```
class K {
    let v : Int
    init(_ v: Int) { self.v = v }
    deinit {
        print("deinit \(v)")
    }
}
var k = K(1)
k = K(2) // 1 n’est plus référencé
k = K(3) // 2 n’est plus référencé
produit :
deinit 1
deinit 2
```

Swift - le chaînage d'optionnels

Ceci permet de remplacer les tests en cascade sur les optionnels :

```
class Node {
    var next : Node?
    var value : Int
    init(_ v: Int) { value = v }
    func set(next n: Node) { next = n }
}

let n1 = Node(1); let n2 = Node(2); let n3 = Node(3)
n1.set(next: n2); n2.set(next: n3)

if let value = n1.next?.next?.next?.next?.value { print("Found \(value)") }
else { print("bad")

if let value = n1.next?.value { print("Found \(value)") }
else { print("bad") }

n1.next?.next?.next?.next?.value = 10
```

Swift - le forçage de type

Le forçage de type (type casting) permet de :

déterminer si un objet est d'un type donné

réaliser un forçage descendant (downcasting)

Le test de type

```
protocol P {}
class K1 : P {}
class K2 : P {}
let t : [P] = [K1(), K2(), K1(), K1()]
for k in t {
    if k is K1 { print("K1")}
    if k is K2 { print("K2")}
}
```

Swift - le forçage de type

Le downcasting peut-être obtenu par
as?
afin d'obtenir un optionnel
ou
as! pour forcer le downcast

```
let p : P = K1()
var k = p as? K2
print(k) // nil
k = p as! K2 // runtime Could not cast value of type…
```

Il ne faut utiliser as! que si on est vraiment sûr de soi…
La forme faible est le plus souvent utilisée avec le if-let

```
if let k1 = p as? K1 { … } else { … }
```

Swift - le type inconnu

Il existe deux types permettant de représenter une référence de type inconnu (fonctionne même s'il n'y a pas de classe de base commune) :

Any

qui représente n'importe quel objet, y compris les fonctions

AnyObject

qui représente n'importe quel objet

```
let t : [Any] = [K(), { (x:Int) in x+1 }, NSObject(), SK()]
```

Swift - les extensions

Les extensions Swift permettent d'étendre n'importe quel type classe, structure, énumération ou protocol.

Y compris si on en possède pas le code source!

Par exemple les entiers ne possèdent pas de méthode permettant de déterminer s'ils sont pairs ou impairs, il suffit de la rajouter :

```
extension Int {
    func isEven() -> Bool { return self%2 == 0 }
}
let i = 20
print(i.isEven())
print(21.isEven())
```

Swift - la gestion mémoire

Dans le cas de cycles de références, le GC est incapable de recycler les objets :

```
class K {
    var other : K? = nil
    deinit { print("deinit") }
}

var k : K? = K()
k = nil
k = K()
k?.other = K()
k?.other?.other = k
k = nil // aucun des deux objets n’est libéré!!!
```

Par défaut, les références sont fortes (strong)

Swift - la gestion mémoire

Pour éviter le problème on peut utiliser une référence faible (weak), ou sinon s'occuper soi-même de briser le cycle :

```
class K {
    weak var other : K? = nil
    deinit { print("deinit") }
}
```

Lorsqu'un objet référencé par une référence faible disparaît, la référence est repositionnée à nil. Les références faibles sont nécessairement optionnelles

```
var k : K? = K()
var o : K? = K()
k?.other = o
print(k?.other)
o = nil
print(k?.other)
```

Swift - la gestion mémoire

Dans le cas d'une relation asymétrique on peut briser la chaîne avec une référence sans propriété (unowned).

Les références unowned ne sont jamais optionnelles.

Par exemple, une classe User représente un utilisateur qui peut posséder des fichiers et une classe File qui représente des fichiers qui peuvent être la possession d'un utilisateur.

Dans ce cas, la disparition d'un utilisateur provoque la disparition de fichiers qui lui appartiennent mais pas l'inverse évidemment!

Swift - la gestion mémoire

```
class User {
    let name : String = "Jean"
    var files = [File]()
    deinit { print("Freeing user \(name)") }
}
class File {
    let name : String
    unowned let user : User
    init(name n:String,user u: User) { name = n; user = u; user.files.append(self) }
    deinit { print("Freeing file \(name)") }
}
var jean : User? = User()
let _ = File(name: "f1", user: jean!)
let _ = File(name: "f2", user: jean!)
let _ = jean?.files.remove(at: 0)
print("Files"); for f in jean!.files { print(f.name) }; print("-Files")
jean = nil
produit :
Freeing file f1
Files
f2
-Files
Freeing user Jean
Freeing file f2
```