Swift 2.2

Jean-Baptiste.Yunes@univ-paris-diderot.fr

2015 v0.1

- Swift (maintenant 2.0)
  - type-safe
  - object-oriented
  - type inference
  - genericity

- Les types de base :
  - Int, Float, Double, Bool, String
  - Array, Set, Dictionnary
  - tuples
- Les types optionnels

- Variables et constantes
  - associent un nom à une valeur d’un type
- Déclarations
  - var anneeCourante = 2016
  - let naissanceTuring = 1912
  - var pi=3.14, e=2.718

- Les déclarations peuvent être annotées d'un type pour assurer celui-ci
  - let zero : Float = 0
- Unicode peut être employé
  - let 👍 = 1, 👎 = -1, 🍟 = "Frites"
  - les flèches, symboles mathématiques ou semi-graphiques sont interdits

- La modification d’une variable s’effectue *via* l’affectation

- var count = 12
  count = 13

- La fonction print permet d’afficher
  - var twoPowerTen = 1024
    print(twoPowerTen)
- Pas de format, il suffit d’utiliser les placeholders
  - print("La valeur est \(twoPowerTen).")

- Les entiers
  - Int8, Int16, Int32, Int64
  - UInt8, UInt16, Uln32, UInt64
- Les bornes
  - Int8.min, Int8.max, etc
- Attention swift est fortement typé, *i.e.* pas de conversion implicite!
- Il est préférable d'utiliser Int, le type des entiers natifs (32-bits ou 64-bits), il existe aussi UInt (éviter de l'utiliser).

- Les littéraux peuvent utiliser
  - la base 10, 945
  - la base 16, 0x4A
  - la base 8, 0o74
  - la base 2, 0b011101

- Swift interdit les dépassements de capacité…
  - une exception est levée
    - var x : UInt8 = 127
      x++ // erreur
- let xx : UInt16 = 2000
  let yy : UInt8 = 20
  let zz = xx + UInt16(yy) // conversion explicite!

- Les alias de type
  - typealias Hertz = Int
- permettent de mieux contextualiser les déclarations

- Les booléens
- Deux littéraux true et false
- C’est le type des expressions de test dans les structures de contrôle :
  - let b = true
    if b { … } else { … }

- Les tuples
  - permettent de renvoyer plusieurs valeurs en retour d'une fonction (par exemple)
  - let monTuple = (500,"Internal Server Error")
    print("Code \(monTuple.0) Message \(monTuple.1)")
  - On peut nommer les éléments du tuple
    let unAutre = (code:404,msg:"Not Found")
    print("Code \(unAutre.code) Message \(unAutre.msg)")

- Les types optionnels
  - sont très utiles car ils permettent une écriture concise du code et garantissent la sûreté
- Une valeur d’un type optionnel c’est
  - ou bien il y a bien une valeur et c’est la valeur
  - ou bien il n’y a pas de valeur

- let mystère = "42"
  let i = Int(mystère)

- L'expression Int(mystère) ne peut garantir qu'une valeur soit produite :

  - let mystère = "DooBeDooBeDoo"
    let i = Int(mystère)

- La conversion produit un Int optionnel, le type correspondant est Int?

  - la constante qui représente «pas de valeur» est nil

  - Attention, nil ne correspond pas au NULL du C, null du Java, etc. Ce n'est pas une constante de pointeur… Il n'y a pas de pointeurs en swift!

- On peut tester les optionnels :
  - if i != nil { print("Ok"); }
- On ne peut pas accéder ordinairement à la valeur
- Si on sait qu'il y a bien une valeur, on peut demander à l'obtenir avec !
  - if i != nil { print("i vaut \(i!)")
  - Attention, il faut être sûr de soi…

- La liaison optionnelle
- L’idiome d’utilisation des optionnels est
  - if let valeur = Int(mystère) {
        print("\(mystère) a été convertit en \(valeur)")
    } else {
        print("Bad \(mystère)")
    }
- On peut aussi utiliser la forme let var…

- On peut combiner la liaison optionnelle et une conditionnelle
  - if let x = …, y = … where x+y>100 { … } else { … }
- Il existe aussi les types optionnels implicitement déballés, ils fonctionnent comme des optionnels mais évitent l'opérateur !. Attention…
  - let i : Int! = nil
    let j : Int = i // runtime error
  - Éviter de les utiliser tant que faire se peut…

- Capture des exceptions :
- ```
  do {
        saut()
        try salto()
        saut()
  } catch GymnastiqueErreur.CotesBrisées {
      print("viré du club")
  } catch {           // default case
      print("oups")
  }
  ```
  ```

- Les assertions
  - permettent d’assurer la continuité du code si une condition est vérifiée, sinon un arrêt de l’application est provoqué

- assert(age<120,"Top vieux")

- Les opérateurs
  - en gros ceux du C
    - note : le modulo est utilisable sur les flottants
  - l’opérateur de coalescence
    - a ?? b
      - équivalent à a != nil ? a : b
- opérateurs de bornage
- 1…10 toutes les valeurs de 1 à 10
- 1..<10 toutes les valeurs de 1 à 9

- Chaînes et caractères
  - les chaînes déclarées avec let sont non mutables
  - le champ characters permet l’itération
  - for i in "😞😟😨👹😇".characters {
    print(i)
    }

- On peut construire une chaîne avec un tableau de caractères
- let tab = ["😞","😟","😨","👹","😇"]
  let chaine = String(tab)
- Note un littéral de Character est une String de longueur 1!
- \ est le caractère d’échappement
- \(…) est un emplacement pour expression

- Chaînes et caractères
  - Le type des indices est Character.Index
  - String.startIndex, String.endIndex (après le dernier)
  - predecessor(), successor(), advanceBy(_:Int)
  - for i in String.characters.indices { … }

- Chaînes et caractères
- Différentes méthodes de manipulation
  - insert, insertContentOf
  - removeAtIndex, removeRange…
- Opérateurs de comparaison
  - =, !=

- Les collections
  - Set,
  - Array,
  - Dictionary (génériques)
- Des bridges vers les types NSArray, NSSet et NSDictionary

- Array
  - Array<*Element*> ou [*Element*] (la seconde est la forme préférable)
  - var tableau = [Int]() // vide
    tableau.append(666) // 1 élément de plus
    tableau = [] // vide
  - var tableau = [Float](count:3, repeatedValue: 666)
  - concaténation +

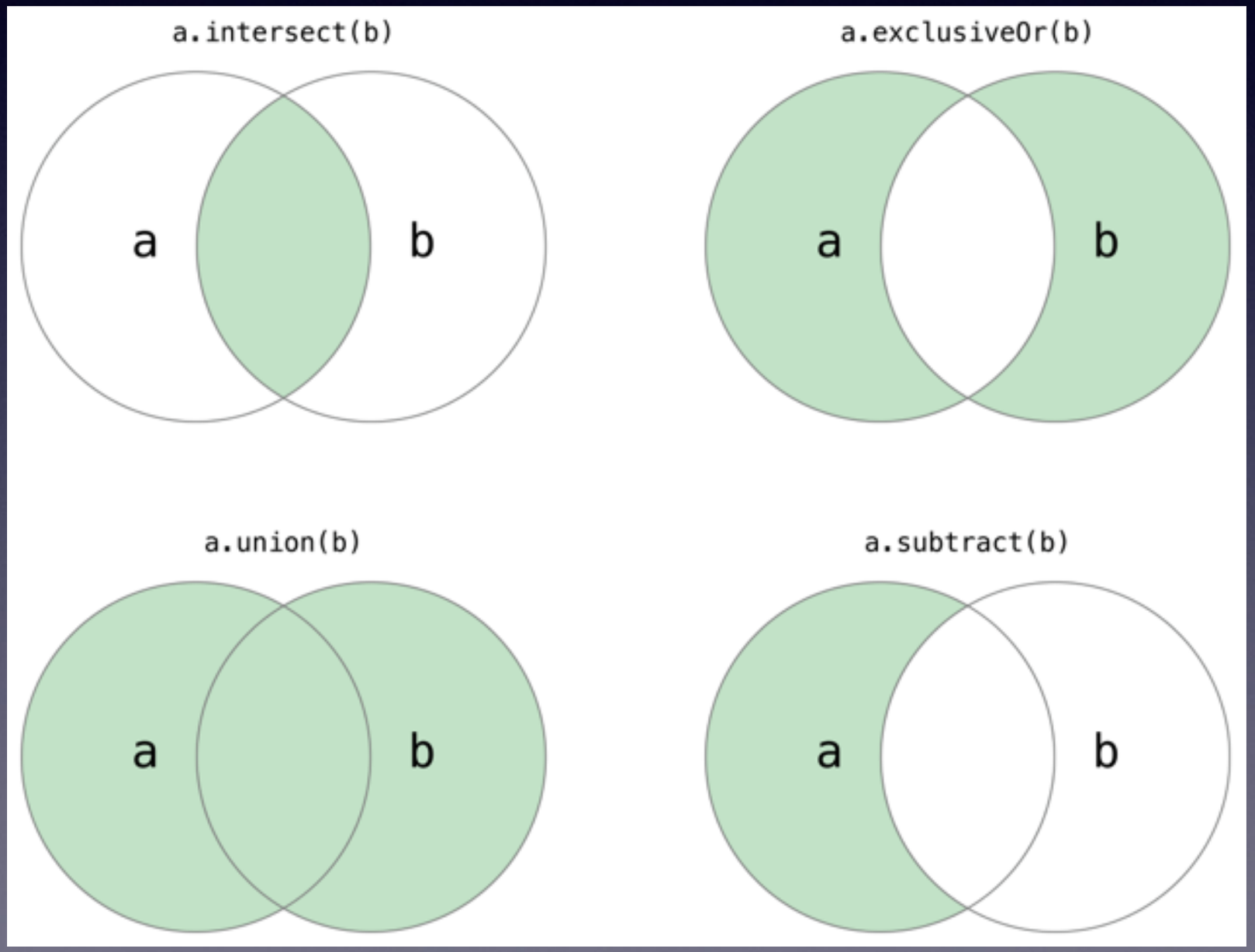
- var cours = ["POCA", "MacOSX"]
  let nCours = cours.count // 2
  if cours.isEmpty { … } // eq. cours.count=0
  cours.append("Internet") // eq. cours +=
  print(cours[0])
  cours[1…1] = ["Cobol expert","Fortran avancé"]
  cours.insert("Rubik’s cube",atIndex:2);
  cours.removeAtIndex(1)
  cours.removeLast()
  for i in cours {
    print("\(i) c’est formidable!")
  }

- Set
- Les éléments doivent être Hashable, ce que sont Int, String, Double, Bool
  - Set<*Element*>

- var lettresUtilisées : Set<Character>()
  lettresUtilisées.insert("a")
  var accents : Set<Character>() = ["é","è"]
  var chiffres : Set = ["0","1","2"] // inférence

- Set
- count
- insert(), remove()
- for l in lettresAccentuées {
  print("\(i) a été utilisée")
  }
- .sort() itérateur ordonné

- Set
- opérations ensemblistes



- ==, !=
- isSubsetOf, isSupersetOf, isStrictSubsetOf, isStrictSupersetOf, isDisjointWith

- Dictionary
- Dictionary<*Key*, *Value*> ou [*Key* : *Value*]
- var aéroports = ["OSA": "Osaka",
  "OZZ" : "Ouarzazate",
  "APL" : "A Programming Language"]
  var naissance = ["Turing":1912, "Einstein":1879]

- Dictionary
- count, isEmpty
- aéroports["OZZ"] = "Ouarzazate International Airport"
if let old = aéroports.updateValue("Aéroport du Kansai", forKey:"OSA") { … }
if let v = aéroports.removeValueForKey("APL") { … }

- Dictionnaires et tuples
- itérations
- for (code,nom) in aéroports {
  print("Le code de \(nom) c’est \(code)")
  }
  for code in aéroports.keys { … }
  for nom in aéroports.values { … }
- let codes = [Array](aéroports.keys)

- Contrôle du flux d’exécution
- boucles : for, while, repeat-while, for-in
- branchements conditionnels : if, guard, switch
- branchements inconditionnels : break, continue, fallthrough, return, throw

- for var i = 0; i<5; i++ { print(i) }
- for i in 0…5 { print(i) }
- Si la variable de contrôle n’a pas d’usage on peut utiliser _ :
  for _ in 1…10 { print("guili") }

- while

- var s = 0, i = 0
  while s < 100 {
    i++; i *= 2
    s += i
  }

- repeat-while

- repeat {
    …
  } while *condition*

- if
- if aéroports.count > 100 { print("Waouh!") }
  else { print("Nazebroque") }

- switch

- let fruit = "Banane"
  switch fruit {
  case "Banane":
    print("Bananier")
  case "Orange":
    print("Oranger")
  default:
    print("Nimportekoiyer")
  }

- Le switch doit être exhaustif, donc un default (en général)…

- Pas de break nécessaire…

- switch et intervalles

- switch i {
  case 0…9:
    print("Quelques unités")
  case 10..<100:
    print("Quelques dizaines")
  default:
    print("Beaucoup")
  }

- … intervalle fermé, ..< intervalle ouvert à droite

- switch matching on tuples

- let quidam=("Yunès","Jean-Baptiste")
  switch quidam {
  case ("Yunès","Jean-Baptiste"):
    print("Un gars zarbi")
  case (_,"Jean-Baptiste"):
    print("Chouette prénom")
  case ("Yunès",_):
    print("Nom bizarre?")
  case (_,_):
    print("Une personne normale")
  }

- Attention à l’ordre…

- Switch matching et liaison

- let quidam=("Poquelin","Jean-Baptiste")
  switch quidam {
  case ("Yunès","Jean-Baptiste"):
    print("Un gars zarbi")
  case (let n,"Jean-Baptiste"):
     print("Chouette prénom ce \(n)")
  case ("Yunès",let p):
    print("Nom bizarre et son prénom \(p)")
  case (_,_):
    print("Une personne normale")
  }

- switch et conditionnelles

- let quidam=("Poquelin","Jean-Baptiste")
  switch quidam {
  case (let n,"Jean-Baptiste"):
    print("Chouette prénom ce \(n)")
  case (let n,let p) where n==p:
    print("Quelle idée de s’appeler comme ça")
  case (_,_):
    print("Une personne normale")
  }

- continue, comme en C
- break, comme en C sauf pour le switch
  - sert à sortir d’un cas lorsque nécessaire
    - le switch est exhaustif, il peut être nécessaire de sortir d’un cas, et aucun cas ne peut être vide, il faut une instruction, break en est une
      - rappel dans son fonctionnement normal le cas d’un switch ne requiert pas de break…
- comme en Java, break et continue peuvent utiliser une étiquette pour désigner la boucle à interrompre/reprendre.

- fallthrough, permet d’obtenir le comportement des cas d’un switch du C
- on passe au cas suivant
    - c’est explicite!

- la sortie anticipée, guard
- si la condition est vraie le code qui suit la garde est exécuté, sinon le corps de la guarde
- guard a>=0 else {
  print("Should be positive or nul")
  return
  }
  // do whatever you like i is positive or nul here
- Attention, le corps de la garde doit provoquer la sortie du bloc englobant : return, exit, continue ou throw ou un appel à une fonction qui ne revient jamais (fatalError)

- API dépendence
- if #available(OSX 10.10, iOS 6, *) {
  } else {
  }
- Joue le rôle des #ifdef du C

- Les fonctions
  - La définition utilise le mot-clé func
    - Une fonction définie doit avoir un nom
    - Éventuellement des paramètres
    - Un type de retour

- Ex :

- func unDePlus(value:Int) -> Int {
  return value+1
  }

- La fonction se nomme unDePlus(_:), le caractère _ indique que le premier paramètre est anonyme, *i.e.* un appel s'effectue de la manière suivante

- unDePlus(12)

- S’il y a plusieurs paramètres
- func somme(a:Int, et:Int) {
  return a+et
  }
- Appel :
- somme(12,et:24)
    - Sauf pour le premier argument, lors de l’appel on doit toujours utiliser le nom du paramètre
    - Attention, il y aura des exceptions à cette règle…
- Son nom est somme(_:et:)

- Si l’on ne veut pas renvoyer de valeur, il suffit de ne pas mettre de retour :

- func trouNoir(a:Int) {
  // …
  }

- Attention : techniquement parlant c’est une fonction qui renvoie un Void dont la seule valeur possible est le tuple vide ().

- On peut renvoyer plusieurs valeurs (tuples)
- func magic(a:Int,b:Int) -> (min:Int,max:Int,moyenne:Double) {

```
  let m = (a<b) ? a : b
  let M = (a>b) ? a : b
  let mean = Double(a+b)/2.0
  return (m,M,mean)
}
```

- On peut renvoyer des optionnels (c’est utile!)
- func magic(a:Int,b:Int) -> (min:Int,max:Int)? {
  guard !(a>0 && b>0) else {
  return nil
  }
  return ((a<b) ? a : b, (a>b) ? a : b)
}

- Les paramètres ont un nom externe et un nom interne
  - sans autre précision le nom interne est aussi le nom externe
- func somme(leNombre a:Int, avec b:Int) -> Int {
  return a+b
  }
- somme(leNombre:12,avec:37)
  - Si le premier paramètre possède un nom externe explicite, il doit alors être spécifié à l’appel

- Si l'on veut permettre de ne pas utiliser de nom externe à l'appel, il suffit d'utiliser le nom _
- func somme(a:Int,_ b:Int) {
  return a+b
  }
- somme(5,6)

- On peut spécifier une valeur par défaut pour un paramètre

- func somme(a:Int=0,_ b:Int=0) -> Int {
  return a+b
  }

- somme(), somme(5), somme(5,6) sont des appels valides

- Liste d’arguments variables en nombre

- func somme(nombres:Int…) -> Int {
  var somme = 0
  for i in nombres { somme += i }
  return somme
  }

- somme(5,6,7,8,9), somme() sont des appels valides

- Les paramètres sont, par défaut, considérés comme des constantes (définies par let)
- On peut avoir besoin de variables
- func f(var a:Int) -> Int {
  a += 123 // sinon interdit
  return a
  }
- Attention, c’est toujours un passage par valeur! C’est le statut du paramètre formel qui change dans la fonction c’est tout…

- Le passage par référence, inout
- func f(inout a : Int) {
  a++
  }
- L’appel nécessite l’opérateur &
  - var x = 0
    f(&x)

- Le type des fonctions
- func add(a:Int, _ b:Int) -> Int {
  return a+b
  }
- func mul(x:Int, _ y:Int) -> Int {
  return a*b
  }
- ont toutes deux le type (Int,Int) -> Int
- let f : (Int,Int)->Int = add
  f(5,6)

- Les fonctions sont manipulables au même titre que les autres types…
- fun apply(f:(Int,Int)->Int,_ a:Int,_ b:Int) -> Int {
  return f(a,b)
  }
- apply(add,12,24)
- Le type fonction peut être utilisé en retour…

- Les fonctions imbriquées
    - On peut définir des fonctions à l'intérieur d'autrs fonctions, leur visibilité est celle du bloc qui contient sa définition
- On peut les utiliser partout (si l'on prend soin de les renvoyer en valeur)

- ```
  func grandeMagie(a:Int) -> (Int,Int)->Int {
    let add(a:Int,_ b:Int) { return a+b }
    let mul(a:Int,_ b:Int) { return a+b }
    return a>=0 ? add : mul
  }
  ```
- grandeMagie(12)(4,5)
  ```

- Les fermetures/clôtures (closures)
- c’est une fonction utilisant des variables libres, *i.e.* des variables qui font partie de son environnement
- une fonction *globale* est une clôture (qui ne capture rien)
- une fonction interne est une clôture (qui capture éventuellement une partie de l’environnement dans laquelle elle est définie
- certaines expressions peuvent êtres des clôtures

- Un exemple:
- let gloubiboulga = ["Turing", "von Neumann", "Lovelace", "Ritchie", "Gosling"]

```
func tri(s1:String, _ s2:String) -> Bool {
   return s1 < s2
}
let gloubi = gloubiboulga.sort(tri)
```

- En utilisant une expression de clôture on peut écrire:

- let gloubi = gloubiboulga.sort({ (s1:String,s2:String)->Bool in return s1>s2 })

- Ce qui suit le in peut être une suite d’instructions

- En utilisant une expression de clôture et l’inférence de type, on peut écrire:

- let gloubi = gloubiboulga.sort({ s1,s2 in return s1>s2 })

- Swift devine que les arguments sont des String et le retour un Bool (c’est ce qui est attendu par sort)

- S’il n’y qu’une seule instruction on peut omettre le return!

- let gloubi = gloubiboulga.sort({ s1,s2 in s1>s2 })

- Encore plus court si on utilise le nommage raccourci des arguments :

- let gloubi = gloubiboulga.sort({ $0>$1})

- Bien mieux encore! Les operateurs sont des fonctions, par conséquent des clôtures, donc
- let gloubi = gloubiboulga.sort(>)

- La variante clôture de traîne: si la clôture est le dernier argument, on peut ne pas la passer (syntaxiquement) comme argument mais la définir comme traîne:

- let gloubi = gloubiboulga.sort() { $0 > $1 }

- Si la fonction ne prend pas d’autres arguments, on peut omettre les parenthèses d’appel

- let gloubi = gloubiboulga.sort { $0 > $1 }

- La capture de valeurs

- ```
  func makeConvertor(rate:Float) -> (Float) -> Float {
    func convertor(value:Float) -> Float {
       return value*rate;
    }
    return convertor
  }
  let USDollarToEuro = makeConvertor(0.93)
  let euro = USDollarToEuro(500)
  let francBurundaisToUSDollar = makeConvertor(1567)
  let dollar = francBurundaisToUSDollar(100)
  ```
  ```

- La variable rate a été **capturée**, elle ne fait pas proprement partie de la fonction qui convertit mais de son environnement
- Les variables capturées peuvent être modifiées si elles ne correspondent pas à des constantes

- Exemple du calcul d'intérêts composés:

- ```
  func makeIntérêtsComposés(var initialRate r:Float,
                                        initialDeposit:Float) -> () -> Float {
      r = 1+r
      func composer() -> Float {
        let v = initialDeposit*r
        r *= r
        return v
      }
      return composer
  }
  let monLivretA = makeIntérêtsComposés(initialRate: 0.01,
                                                  initialDeposit: 100) // 1%
  let premièreAnnée = monLivretA() // 101
  let deuxièmeAnnée = monLivretA() // 102.01
  let troisièmeAnnée = monLivretA() // 104.0604
  ```
  ```

- Le non-échappement

- Si une clôture reçue en argument n’est pas utilisée pour *ressortir* elle peut être qualifiée de @noescape (afin d’obtenir les bonnes optimisations)

- Autre «bizarrerie» : les @autoclosure et @autoclosure(escaping)

    - sachez que cela existe, lorsque vousles rencontrerez il sera temps de se renseigner<

- Enumérations
- Ce sont des véritables types…
- enum Jours {
  case Lundi
  case Mardi
  case Mercredi
  case Jeudi
  case Vendredi
  case Samedi
  case Dimanche
  }
- ou enum Jours {
  case Lundi,Mardi,Mercredi,Jeudi,Vendredi,Samedi,Dimanche
  }

- let jourCourant = Jours.Lundi
  print(jourCourant)
  jourCourant = .Mercredi // jourCourant déjà typé

- switch jourCourant {
    case .Lundi:
      print("Début de la semaine")
    case .Samedi:
      print("Le week end!")
    default:
      print("Un jour ordinaire")
  }

- Les cas d’une énumération peuvent être typés pour leur associer des valeurs:

- ```
  enum Carburant {
    case SP95,SP95E10,SP98,Diesel
  }
  enum Vehicule {
    case Voiture(carburant:Carburant,places:Int)
    case Camion(volume:Int)
  }
  let maVoiture =
  Vehicule.Voiture(carburant:Carburant.Diesel,
                                  places:5)
  ```
  ```

- On peut alors capturer les valeurs:

- ```
  switch unVehicule {
    case .Voiture(let car,let pla):
      print("Une voiture qui roule au \(car) et de \(pla) places")
    default:
      print("…")
  }
  ```

- On peut aussi l’écrire:

- ```
  switch unVehicule {
    case let .Voiture(car,pla):
      print("Une voiture qui roule au \(car) et de \(pla) places")
    default:
      print("…")
  }
  ```

- Une énumération peut être typé afin d'associer les cas à une valeur (uniforme)

- enum Separator : Character {
  case Tabulation = "\t"
  case RetourChariot = "\n"
  case Espace = " "
  }

- ou implicitement:

- enum Nombres : Int {
  case zero = 0,un,deux,trois,quatre
  }

- Le cas particulier des énumérations du sous-type String

- Les cas auront par défaut la chaîne correspondant à l'identificateur de cas:

- enum Couleurs: String {
  case Rouge,Bleu,Vert
  case Noir = "Black"
  }

- Pour les énumérations typées, on accède à la valeur à l’aide du champ rawValue
- print(couleur.rawValue)
- On peut aussi utiliser le constructeur (d’un optionnel correspondant)
- let couleur.init(rawValue:"Jaune")
  print(couleur?.rawValue)

- Les énumérations récursives…
- func whichInt(i:Church) -> Int {
  switch i {
  case .Zero:
  return 0
  case .Succ(let pred):
  return whichInt(pred)+1
  }
  }
  whichInt(.Succ(.Succ(.Succ(.Zero))))

- Classes et Structures
- Bien que similaires:
  - les classes permettent de définir des objets
  - les structures définissent des valeurs…
- Les structures sont un peu plus limitées (pas d’héritage, pas de déinitialiseur — kézako?)

- class CompteEnBanque {
  }
- struct Point {
  }
- let cc = CompteEnBanque()
  let p = Point()

- Les classes et structures peuvent posséder des propriétés (variables ou constantes)

- structure Point {
  let x:Int, y:Int
  }

- Swift génère automatiquement un initialiseur

- let origine = Point(x:0,y:0)

- Attention : en Swift on ne parle pas de constructeur… On parle d'initialiseur… Un initialiseur sert à initialiser!

- Attention
  - les structures et les énumérations définissent des valeurs! (⟹ passage par valeur)
  - les classes définissent des références vers des objets! (⟹ passage par référence)

- Opérateurs d’identité
  - === (trois =) est l’opérateur qui teste si deux références désignent le même objet
  - !== est l’opérateur qui teste si deux références désignent chacune un objet différent

- Les types standards String, Array, Dictionary ?
  - ce sont des structures!
- Attention : car les NSString, NSArray et NSDictionary de Foundation sont des classes…

- Les propriétés sont de deux genres :
  - les propriétés stockées (stored properties)
    - uniquement pour les classes et structures
  - les propriétés calculées (computed properties)
- Les propriétés sont généralement attachées aux instances du type (objets ou valeurs)
- On utilise plus rarement les propriétés de types (type properties)

- Les propriétés stockées
  - rien de surprenant (pour une utilisation de base)
- struct Personne {
  let id : Int
  var nom : String
  }
  let jby = Personne(id:666,nom:"Yunès")

- Les propriétés stockées fainéantes (kézako?)
- Dans certains cas, il peut être considéré comme inutile que les propriétés aient toutes des valeurs définies au moment ou l'instance est créée
    - l'important est que les valeurs soient définies au moment où l'on les utilisent
    - le nom de la personne peut être recherché dans une base de données par exemple
        - par conséquent si on utilise jamais le champ nom, inutile de consulter la base

- func dbGetNom(id:Int) -> String {
  …
  }
  struct Personne {
  let id : Int
  lazy var nom : String = { dbGetNom(self.id) }()
  }

- Nécessite un initialiseur…

- Les propriétés calculées
  - utiles dans le cas où une propriété peut être déduite (par calcul) de la valeur d'autre propriétés
- struct Cercle {
  let x: Float, y: Float, rayon: Float
  var diametre : Float {
  get { // le getter
  return 2*rayon
  }
  }
}
let unCercle = Cercle(x: 5, y: 12, rayon: 10)
print(unCercle.diametre)

- L’exemple est celui d’une propriété en lecture seule (read-only property)
  - il existe une version «courte»
- struct Cercle {

```
  let x: Float, y: Float, rayon: Float
  var diametre : Float {
     return 2*rayon
  }
}
let unCercle = Cercle(x: 5, y: 12, rayon: 10)
print(unCercle.diametre)
```

- On peut aussi avoir un setter…

- ```
  struct Temps {
   var secondes : Float = 0
      var minutes : Float {
          get {
              return 60/self.secondes
          }
          set {                                          // paramètre implicite
              self.secondes = newValue*60
          }
      }
      var heures : Float {
          get {
              return self.minutes/60
          }
          set(v) {                                       // paramètre nommé explicite parameter
              self.minutes = newValue*60
          }
      }
  }
  ```
  ```

- On remarquera l’emploi de la variable implicite newValue, mais on peut employer un paramètre nommé si l’on préfère

- Cocoa fait un emploi important du pattern Observer
- C’est donc inclus dans Swift pour les propriétés…

```
struct Temps {
    var secondes : Float = 0 {
        willSet {
            print("Ancienne valeur \(secondes) nouvelle \(newValue)")
        }
        didSet {
            print("Ancienne valeur \(oldValue) Nouvelle valeur \(secondes))")
        }
    }
    var minutes : Float {
        get {
            return 60/self.secondes
        }
        set {
            self.secondes = newValue*60
        }
    }
    var heures : Float {
        get {
            return self.minutes/60
        }
        set(v) {
            self.minutes = v*60
        }
    }
}
var temps : Temps = Temps(secondes: 45)
print(temps.secondes)
temps.secondes = 46
print(temps.secondes)
temps.heures = 2
```

- On remarquera l’emploi de oldValue dans le post-observateur
- Toute variable (pas seulement les propriétés) peuvent être calculées…
  - Une variable qui ne change jamais (un exemple zarbi)
- var v : Float {
  get { return 3.0 }
  set { }
  }
  print(v)
  v = 5
  print(v)

- Les propriétés de types sont déclarées avec l’attribut static
- class MyClass {
  static var instanceCount : Int
  }

- Les méthodes
  - il y a des méthodes d’instance
  - et des méthodes de type
- class Compteur {

```
  var value : Int = 0
  func inc() {
    value++
  }
}
var compteur = Compteur()
compteur.inc()
```

- Pour les types qui définissent des valeurs, on peut modifier la valeur à l'aide de méthodes qualifiées de mutating

- struct Wrap {
  var v : Int = 0
  mutating func m() { self.v++ }
}
var w = Wrap()
w.m()

- On peut, plus radicalement, modifier self lui-même! :

- struct Wrap {
  var v : Int = 0
  mutating func m() { self = Wrap(v:self.v+1) }
  }
  var w = Wrap()
  w.m()

- rien de surprenant (!?!), il s’agit de valeurs puisque des structures…

- On peut évidemment surcharger toute méthode

- Les méthodes de type (méthodes statiques dans d’autres languages) sont qualifiées à l’aide du mot-clé class…
- à l’intérieur d’une telle méthode, le mot-clé self désigne la classe elle-même

- L’indexation des classes, structures et énumeration
- Une instance peut-être vu comme un conteneur dont les éléments peuvent être manipulés *via* la syntaxe d’indexation []
- Il suffit de définir l’opérateur [] *via* la définition de la propriété subscript

```
struct Primes {
    subscript(idx: Int) -> Int {
        get {
            switch(idx) {
            case 1: return 1
            case 2: return 2
            case 3: return 3
            case 4: return 5
            case 5: return 7
            case 6: return 11
            case 7: return 13
            default: return -1
            }
        }
    }
}

var p = Primes()
print(p[2])
print(p[3])
```

- La propriété subscript peut définir le getter et le setter
  - Si le setter n'est pas défini, on peut alors utiliser la forme courte pour le getter
  - struct Primes {

```
    subscript(idx: Int) -> Int {
        switch(idx) {
          case 1: return 1
          case 2: return 2
          default: return -1
        }
    }
```

- La définition d’une indexation n’est pas limitée au type Int
  - n’importe quel type peut être utilisé pour l’indexation
  - un nombre quelconque d’arguments peut être utilisé pour l’indexation (multidimensionnelle)

```
struct ToString {
    subscript(word:String) -> Int {
        if word=="un" { return 1 }
        return Int.max
    }
}

let t = ToString()
print(t["un"])
```

```
struct Multiplication {
    subscript(plieur:Int, plicande:Int) -> Int {
        return plieur*plicande;
    }
}

var tm = Multiplication()
print(tm[5,6])
```

- L’héritage
  - est réservé aux classes
  - autorise la redéfinition de méthodes, propriétés et indexations
  - autorise la définition de nouvelles méthodes, propriétés et indexations
- Attention : il n’y a pas de classe de base par défaut en Swift…

```
class ObjetColoré {
    let couleur : (Int,Int,Int)
    init(_ r: Int,_ v: Int,_ b: Int) {
        couleur = (r,v,b)
    }
}

class FigurePleine : ObjetColoré {
    let position : (Int,Int)
    init( p: (Int,Int), c: (Int,Int,Int)) {
        position = p
        super.init(c.0, c.1, c.2)
    }
    func toString() -> String {
        return "À la position (\(position.0),\(position .1)) et avec la couleur (\(couleur.0),\
(couleur.1),\(couleur.2))"
    }
}

let fp = FigurePleine(p: (1, 2), c: (3,4,5))
print(fp.toString())
```

- Il est possible d'empêcher la redéfinition *via* le mot-clé final (méthodes, propriétés calculées, ou index)
- Toute redéfinition doit employer le mot-clé override

```
class Base {
    var className : String {
        return "Base"
    }
    final func f() -> String {
        return "In Base.f(), real class \(className)"
    }
    func g() -> String {
        return "In Base.g()"
    }
}
class SousClasse : Base {
    override var className : String {
        return "SousClasse"
    }
    override func g() -> String {
        return "In SousClasse.g()"
    }
}
```

- Les initialisations
  - les propriétés stockées doivent toujours être initialisées
  - l'initialisation doit être effectuée :
    - à l'aide d'une valeur par défaut
    - dans un initialiseur
- Attention : lors des initialisations, les observeurs ne sont pas appelés

```
class Capteur {
    var temperature : Float
    init() {
        temperature = 0.0
    }
}
```

- ou :

```
class Capteur {
    var temperature : Float = 0.0
}
```

- Les initialiseurs peuvent prendre des paramètres :

```
class Capteur {
    var temperature : Float
    init(enCelsius celsius : Float) {
        temperature = celsius
    }
    init(enKelvin kelvin : Float) {
        temperature = kelvin-273.0
    }
}
```

- Seules les propriétés optionnelles sont, sans contre-indication, initialisées à une valeur par défaut : nil

- Les structures qui n’auraient pas défini d’initialiseur, et sans indication contraire, ont un initialiseur synthétisé permettant d’initialiser les champs, c’est l’initialiseur par champ :

```
struct Ordinateur {
    let fréquence : Float
    let mémoire : Int
    let disque : Int64
}

let o = Ordinateur(fréquence: 2.3,
      mémoire: 20000000,
      disque: 500000000000)
```

- La délégation d’initialisation :
    - pour les types valeurs, structures ou énumérations, la règle est simple, on utilise self.init
        - attention, la définition d’un initialiseur pour un type valeur masque l’initialiseur par défaut
    - pour les classes, cela se complique, en particulier avec l’héritage

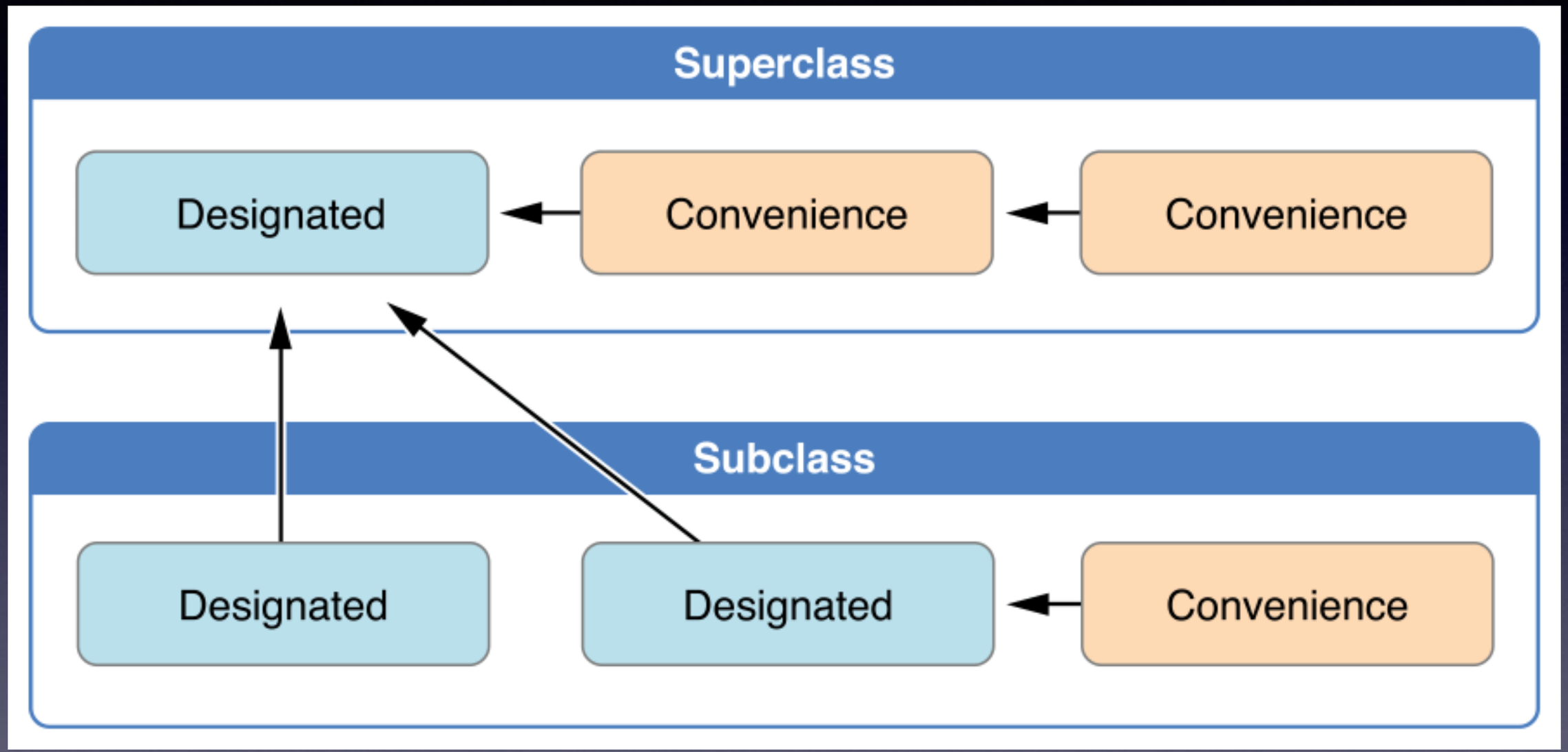
```
struct Point {
    let x : Float
    let y : Float
    init() {
        self.init(x: 0.0,  y: 0.0)
    }
    init(x: Float, y: Float) {
        self.x = x; self.y = y
    }
}
```

```
struct Ligne {
    let origine: Point, extrémité: Point
    init() {
        self.init(p1: Point(), p2: Point())
    }
    init(p1: Point, p2: Point) {
        origine  = p1; extrémité = p2
    }
    init(x1: Float, y1: Float, x2: Float, y2: Float) {
        self.init(p1: Point(x: x1, y: y1), p2: Point(x: x2, y: y2))
    }
}
```

- Pour les classes, il y a deux types d'initialiseurs :
  - les initialiseurs primaires, sont ceux qui contiennent le «véritable» code d'initialisation
  - les initialiseurs utiles, sont ceux qui servent à définir des syntaxes alternatives et pratiques; ils sont marqués par le mot-clé convenience

- Les trois règles d’initialisation pour les classes :

1. les initialiseurs primaires doivent appeler un initialiseur primaire de la super classe
2. les initialiseurs utiles ne peuvent appeler que des initialiseurs de la classe
3. l’appel à un initialiseur utile doit ultimement terminer par un appel à un initialiseur primaire



extrait de l’ouvrage The Swift Programming Language (Swift 2.1), Apple

```
class Point {
    var x: Float
    var y: Float
    init(x: Float, y: Float) {
        self.x = x; self.y = y
    }
    convenience init() {
        self.init(x: 0.0, y: 0.0)
    }
}
```

- L’initialisation comporte essentiellement deux phases :

    1. les propriétés stockées doivent être initialisées par la classe dans laquelle elles apparaissent;

    2. puis le processus continue de sorte que l’initialisation soit complète

- Swift assure les propriétés suivantes :

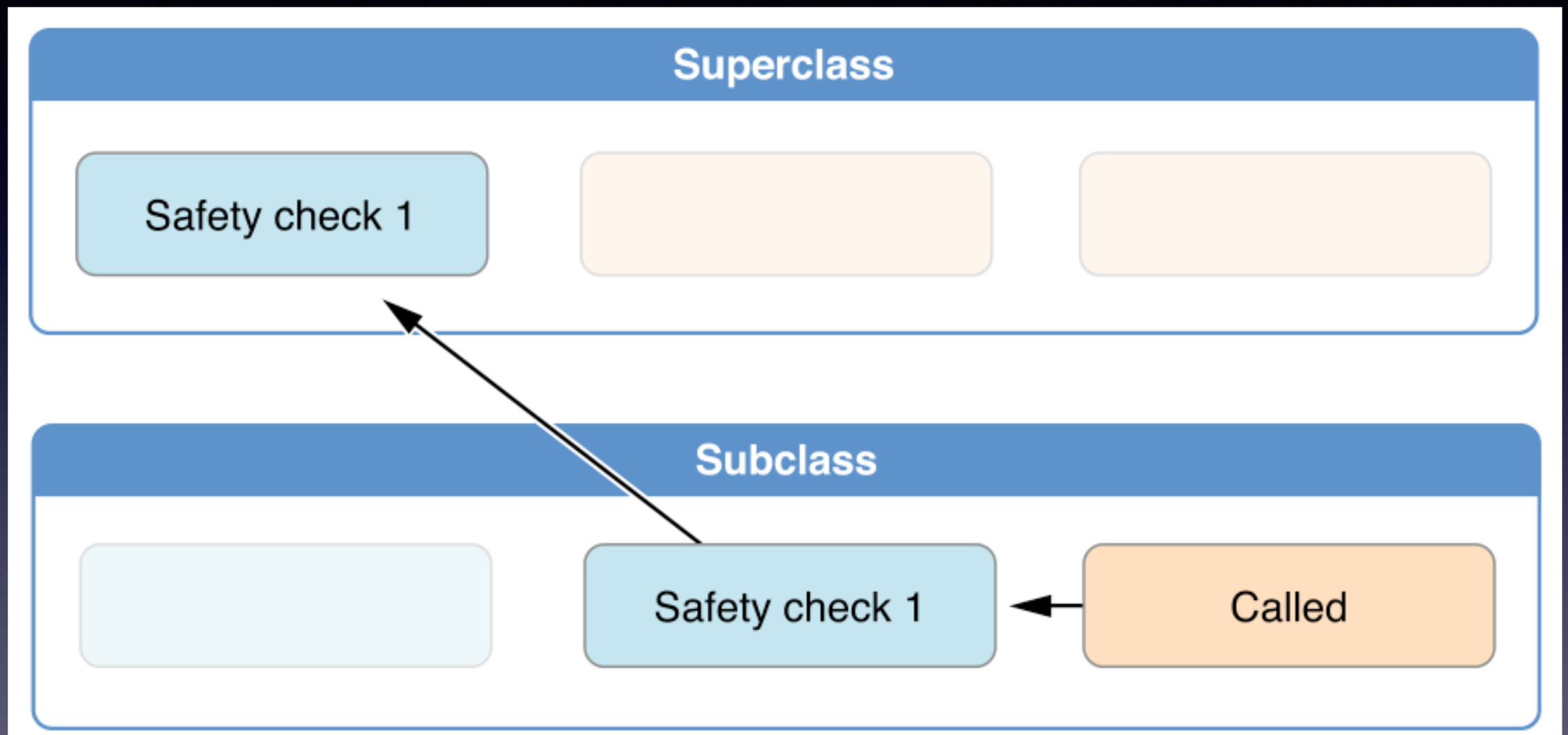
1. un initialiseur primaire doit initialiser les propriétés stockées avant de faire appel à la délégation

2. un initialiseur primaire doit faire appel à la délégation avant d'initialiser une propriété héritée

3. un initialiseur utile doit faire appel à un initialiseur primaire avant d'initialiser une propriété

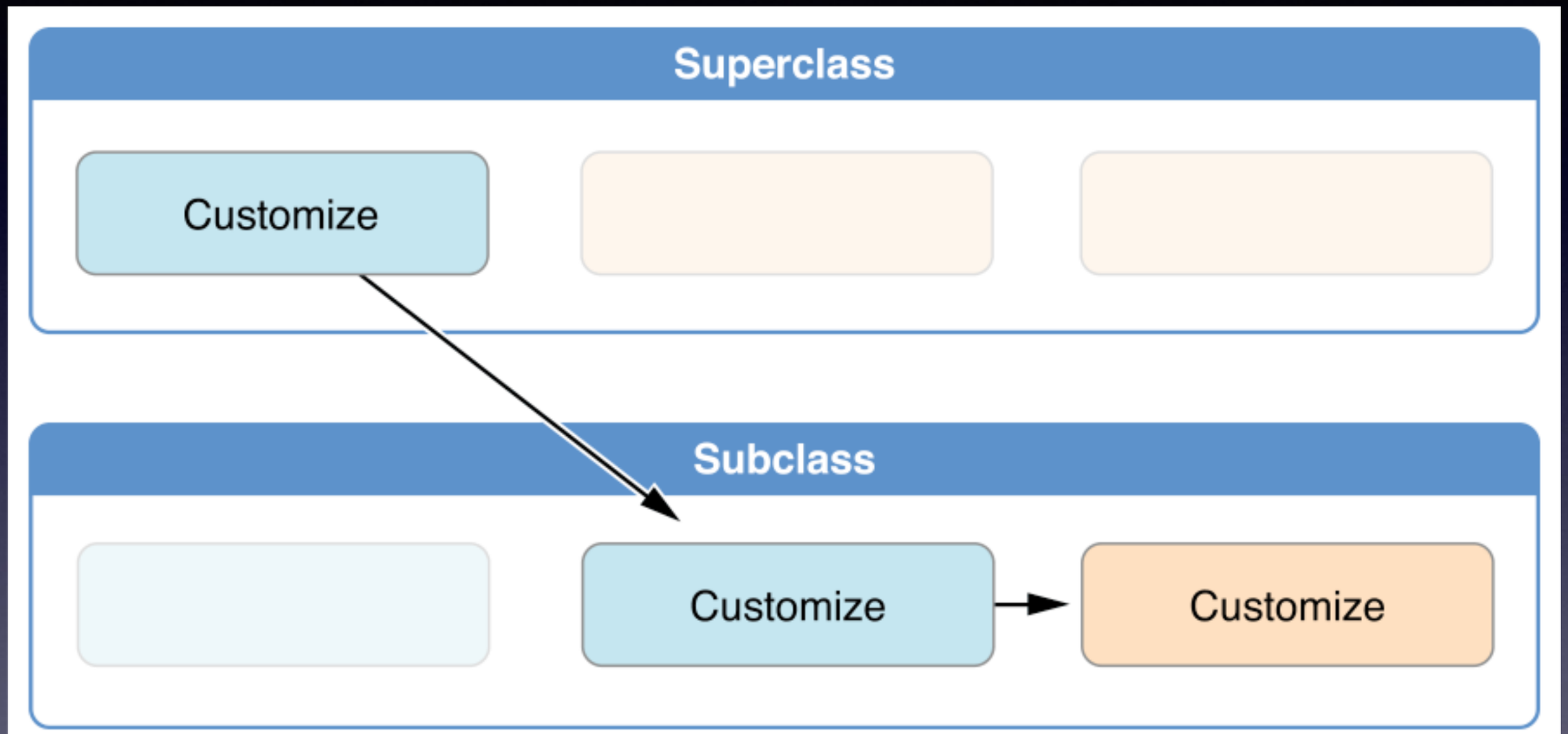
4. aucun initialiseur ne peut faire appel à une méthode, tenter de consulter la valeur d'une propriété ou faire référence à self en tant que valeur tant que la première phase n'est pas complète

- Phase 1
  - un initialiseur est appelée sur la classe
  - la mémoire est allouée mais non initialisée pour l’instance
  - un initialiseur primaire finit par être appelé et assure que les propriétés de la classe sont initialisées
  - l’initialiseur primaire fait appel à la délégation afin d’assurer l’initialisation au niveau supérieur, ceci continuant jusqu’en haut de la chaîne d’héritage. Une fois la racine atteinte, la phase 1 est terminée est toutes les propriétés possèdent une valeur



extrait de l’ouvrage The Swift Programming Language (Swift 2.1), Apple

- Phase 2
  - les appels «redescendent» en permettant à chaque initialiseur d’affiner ses initialisations en autorisant l’utilisation de méthodes, la consultation des propriétés et l’utilisation de self comme valeur
  - les appels aux initialiseurs utiles sont autorisés à affiner l’initialisation…



extrait de l’ouvrage The Swift Programming Language (Swift 2.1), Apple

- Il n’y a pas d’héritage des initialiseurs (sauf certains cas très particuliers)
- La définition d’un initialiseur de même prototype qu’un appartenant à la super-classe nécessite l’emploi du mot-clé override, y compris pour les initialiseurs utiles

- Les initialiseurs qui peuvent échouer :
  - se nomment init?
  - doivent «renvoyer» nil en cas d’échec, attention les initialiseurs ne renvoient pas de valeur…

```
class Individu {
    let nom : String
    init?(nom: String) {
        self.nom = nom // class, safety check 4
        if nom.isEmpty { return nil }
    }
}

if let jb = Individu(nom: "yunès") {
    print("Voici \(jb.nom)")
} else {
    print("raté")
}
```

- Note : on peut aussi définir des initialiseurs qui peuvent échouter mais définissent un type automatiquement déballé, avec la syntaxe init!

- Éviter d’employer les type automatiquement déballés

- Le mot-clé required sert à marquer un initialiseur de sorte que les sous-classes soient obligées de le redéfinir

- On peut utiliser une clôture pour initialiser une propriété :

```
struct BrahmaneSissa {
    let plateau : [[Int]] = {
        var t: [[Int]] = [[Int]]()
        var p = 0
        for i in 0...3 {
            t.append([Int]())
            for j in 0...3 {
                t[i].append(2<<p)
                p++
            }
        }
        return t
    }()
}
```

- La dé-initialisation est l’instant qui précède la désallocation
    - elle peut faire appel à la méthode deinit si elle existe afin d’exécuter un code particulier comme la libération d’une ressource
    - Attention : Swift utilise l’ARC (une sorte de garbage collector) et les désallocations sont «automatisées»

```
struct BanqueMonopoly {
    static var total = 1000000
    static func prendre(montant: Int) -> Int {
       assert(total>=montant)
        total -= montant
        print("Il reste \(total) en banque")
        return montant
    }
    static func rendre(montant: Int) {
        total += montant
        print("Il y a \(total) en banque")
    }
}

class Joueur {
    var compte : Int
    init() { compte = BanqueMonopoly.prendre(20000) }
    deinit { BanqueMonopoly.rendre(compte) }
}

var j1 = Joueur?()
j1 = nil
```

- L’ARC (Automatic Reference Counting)
  - C’est un gestionnaire automatisé de l’allocation mémoire
    - Il repose sur le comptage de référence
      - nécessite donc une attention pour certaines structures de données (circulaires)
  - N’est utilisé que pour les instances de classes…

- Par défaut les références sont «fortes» (strong) :

```
class QuelqueChose {
    init () {
        print("Un \(self.dynamicType) en plus!")
    }
    deinit {
        print("Un \(self.dynamicType) en moins")
    }
}
var r1 : QuelqueChose?
var r2 : QuelqueChose?
r1 = QuelqueChose()
r2 = r1
r1 = nil
r2 = nil
```

```
class Individu {
    let name : String
    init(name: String) {
        self.name = name
        print("\(self.name) naît")
    }
    var friend : Individu? // référence forte!
    func aPourAmi(personne: Individu) {
        friend = personne
        personne.friend = self
        print("\(self.name) a pour ami(e) \(personne.name)")
    }
    deinit { print("\(name) disparaît") }
}

var abhaya: Individu? = Individu(name: "Abhaya") // Sans crainte
var nozomi: Individu? = Individu(name: "Nozomi") // L'espoir

abhaya?.aPourAmi(nozomi!)

abhaya = nil
nozomi = nil // pas de libération des deux individus
```

- Dans ce type de cas (une référence circulaire qui peut ne pas avoir de valeur associée), on peut qualifier la référence de weak (faible)
    - de la sorte la référence ne «compte» pas pour ARC et l'instance peut-être libérée
- et la référence faible est associée à nil
    - une référence faible doit nécessairement désigner un type optionnel…

```
class Individu {
    let name : String
    init(name: String) {
        self.name = name
        print("\(self.name) naît")
    }
    weak var friend : Individu? // weak reference!
    func aPourAmi(personne: Individu) {
        friend = personne; personne.friend = self
    }
    func afficheAmitié() {
        if let ami = friend?.name { print("\(self.name) a pour ami(e) \(ami)") }
        else { print("\(self.name) n'a pas d'ami") }
    }
    deinit { print("\(name) disparaît") }
}

var abhaya: Individu? = Individu(name: "Abhaya") // Sans crainte
var nozomi: Individu? = Individu(name: "Nozomi") // L'espoir

nozomi?.afficheAmitié()
abhaya?.aPourAmi(nozomi!)
nozomi?.afficheAmitié()

abhaya = nil
nozomi?.afficheAmitié()
nozomi = nil
```

- Les références sans possession (unowned) correspondent au cas où la référence doit nécessairement désigner une valeur mais sans qu’elle ne compte pour ARC

- donc pour le cas des références non-optionnelles

```
class CarteIdentité {
    unowned var qui : Personne
    init(personne: Personne) {
        qui = personne
        qui.carteIdentité = self
    }
    func affiche() { print("Carte de \(qui.nom)") }
    deinit { print("Destruction carte") }
}

class Personne {
    var nom: String
    var carteIdentité: CarteIdentité?
    init(nom: String) { self.nom = nom }
    deinit { print("Bye \(nom)") }
}

var jby : Personne? = Personne(nom: "Jean-Baptiste Yunès")
jby?.carteIdentité = CarteIdentité(personne: jby!)
jby?.carteIdentité?.affiche()
jby = nil
```

- le chaînage optionnel (optional chaining)

- Le chaînage optionnel est la possibilité d’enchaîner les accès à des propriétés ou méthodes y compris lorsqu’elles correspondent à des types optionnels
- La valeur nil est propagée et l’expression produit un optionnel
  - self.window?.getView()?.size
  - self.window?.views?[0].size

- La gestion d’erreur
  - les erreurs doivent se conformer au protocole ErrorType
  - on verra plus loin mais les protocoles sont des interfaces au sens de la POO

- la définition d’un type d’erreur

```
enum FileCreationError: ErrorType {
    case BadPath(path: String)
    case BadAccess(access: Int)
}
```

- La notification d’un erreur

throw FileCreationError.BadPath(path:"/bricolo/debilo")

- Deux modes de gestion des erreurs :
  - la propagation
  - la capture

- La propagation doit être spécifiée dans la signature de la fonction :
  func blabla(param: Type) throws -> returnType {}
- une telle fonction est qualifiée de *throwing function*
  - on ne précise pas quels sont les types propagés

- La capture do, try, catch

- try doit qualifier l’appel à toute expression qui pourrait renvoyer une erreur :
  try machin.blabla()

- do catch permet de capturer les erreurs d’un bloc d’expression

- ```
  do {
    let of = OpenedFile("/bricolo/rigolo")
    try of.open()
  } catch FileCreationError.BadPath(let path) {
    print("…")
  } catch { // default case
    print("je sais pas quoi faire…")
  }
  ```

- Si les cas ne sont pas exhaustifs il faut alors propager en marquant la fonction qui contient le bloc comme *throwing function*
  ```

- l’évaluation optionnelle d’expressions avec erreur potentielle

    - try? permet d’obtenir une valeur du type optionnel correspondant :

```
enum IntError : ErrorType {
    case ZeroValue
}
func f(a:Int) throws -> Int {
    if a == 0 { throw IntError.ZeroValue }
    return a+1
}

let a = try? f(0)
print(a)
```

- L’évaluation forcée
  - Si on sait que la fonction fonctionnera à tout coup on peut éviter l’obtention d’un optionnel en utilisant try!
- let a = try! f(1)

- L’exécution différée
  - On peut avoir besoin d’obtenir l’exécution d’un code quelque soit la sortie (return, break, throw)
  - Il suffit pour cela de déclarer un bloc d’exécution différée (*defer block*)
  - On peut avoir autant de blocs d’exécution différée, ils sont simplement exécutés dans l’ordre inverse de leur création

- Le test de type
  let SPEED_OF_LIGHT = 300_000
  class Vehicle {
   var maxSpeed = SPEED_OF_LIGHT
}

class Car : Vehicle {}

class Bicycle : Vehicle {}

let vehicles = [Bicycle(), Car(), Car(), Bicycle()]

var cars = 0
var bicycles = 0
for v in vehicles {
        if v is Car { cars++ }
    else if v is Bicycle { bicycles++ }
}
print("\(cars) cars, \(bicycles) bicycles")

- Downcasting

```
class Bicycle : Vehicle {
 var wheelsDiameter = 27.5
}
print("Diameter \((vehicles[0] as?
Bicycle)?.wheelsDiameter)")
print("Diameter \((vehicles[0] as!
Bicycle).wheelsDiameter)")
print("Diameter \((vehicles[1] as?
Bicycle)?.wheelsDiameter)")
```

- Les types génériques
- AnyObject peut représenter n’importe quelle instance de n’importe quelle classe
- Any peut représenter n’importe quelle instance de n’importe quel type (y compris les fonctions)
- Ces types s’utilisent essentiellement avec l’API Cocoa car Objective-C n’avait pas de tableaux génériquement typés.

```
let porteninwak : [Any] = [4, 5.6, Car(), { (s: String) in return "--\(s)--"}]

for o in porteninwak {
    switch o {
    case let i as Int :
        print("Pas d'intérêt!")
    case let f as Float :
        print("Y'a de la flotte!")
    case let c as Car :
        print("Caramba!")
    case let f as String -> String :
        print(f("Dobedo"))
    default :
        print("Je suis pris en défaut")
    }
}
```

- Les types imbriqués
  - Swift permet d'imbriquer la définition de types

```
class JeuDeBelote {
enum Couleur : Character {
    case Pique = "♠"
    case Cœur = "♥"
    case Carreau = "♦"
    case Trêfle = "♣"
    }
}
print(JeuDeBelote.Couleur.Cœur.rawValue)
```

- La rétro-modélisation (retro-modeling) par les extensions
- Les extensions permettent d’enrichir une classe quand bien même on ne possède pas l’accès à son code!
- On peut rajouter des propriétés(calculées!), des méthodes, des initialiseurs…

- Le type Int *manque* de fonctionnalités ?
  Rajoutez-en!

```
extension Int {
    func isEven() -> Bool { return self%2 == 0 }
    func isOdd() -> Bool { return !isEven() }
}

let x = 13
print(x.isOdd())
```

- Avec les propriétés calculées on obtient par exemple :

```
extension Double {
    var $ : Double { return self }
    var €: Double { return self/0.897686214 }
}

var USAccount = 3.5.$
var frenchAccount = 45.€
```

- Extensions mutantes :

```
extension Int {
    mutating func double() { self = 2*self }
}
var i = 14
i.double()
print(i)
```

```
extension Int {
    subscript(i: UInt) -> Int {
        return self*Int(i)
    }
}

print(3[4])
```

- une extension peut aussi permettre d’ajouter la conformité à un Protocole (voir plus loin…)
- extension Int : Printable {
  // implements Printable methods
  }

- Les protocoles
- Il s’agit de la notion d’interface, on dit
  - qu’une classe, une structure ou une énumération adopte un protocole
  - qu’un type qui satisfait un protocole est conforme au protocole
  - protocol Printable {
    // définition du protocole
    }

- Les propriétés étant (d'une certaine manière) unifiées avec les méthodes on peut imposer des propriétés par un protocole :

- protocol Chelou {
  var readWrite : Int { get set }
  var readOnly : Int { get }
  }

- protocol Identifié {
  var id : Int { get }
  }
  class Véhicule : Identifié {
  static var serial = 1
  let id : Int
  init() { id = Véhicule.serial++ }
  }

- Les méthodes de protocole
- protocol Printable {
  func toString() -> String
  }

- Les méthodes de protocole peuvent être marquées mutating lorsque la sémantique de la méthode correspond à la modification de l'état des objets

- protocol Invertible {
  mutating func invert()
}

```
extension Int: Invertible {
    mutating func invert() { self = -self }
}

var xx = 3
xx.invert()
```

- Un protocole peut aussi imposer des initialiseurs
- protocol InitializableWithInt {
  init(v: Int)
}
class K: InitializableWithInt {
  required init(v: Int) { Swift.print("Yes!") }
}

- Les protocoles peuvent être des spécialisations de protocoles :
- protocol PrettyPrintable: Printable {
  //
  }

- Si la sémantique du protocole correspond à des références et non à des valeurs, alors on peut le restreindre aux classes

- protocol P: class {
  //
  }

- Les protocoles peuvent être composés pour imposer l'adoption multiple (en évitant la définition d'un nouveau protocole *ad-hoc*) :

- func f(v: protocol<Imprimable,Zoomable>)

- Il est possible d’obtenir des protocoles faibles, c’est-à-dire qui n’imposent pas d’implémenter certaines méthodes du protocole

  - celui-ci doit être marqué @objc

- ```
  @objc protocol Zarbi {
    optional func f() -> Void
    optional func g() -> Void
  }
  ```

- Seules les classes héritant conformes à l’API Cocoa peuvent les utiliser (NSObject…)
  ```

- Les protocoles peuvent être étendus (par extension) pour :
  - ajouter des fonctionnalités à tous les types conformes
  - fournir des implémentations par défaut des méthodes du protocole

- La généricité

- Une méthode générique pour échanger deux variables d'un type quelconque :

```
func mySwap<T>(inout a: T, inout withB b: T) {
    let t = a
    a = b
    b = t
}

var a1 = 2
var a2 = 3
print("\(a1) \(a2)")
mySwap(&a1, withB: &a2)
print("\(a1) \(a2)")
```

- Une classe générique :

```
class Bag<Element> {
    var elements = [Element]()
    func add(item: Element) -> Void {
        elements.append(item)
    }
    func isEmpty() -> Bool {
      return elements.count == 0
    }
}
```

- On peut étendre un type générique
  - L’extension ne fait pas apparaître le type dans sa définition mais peut l’utiliser :

```
    extension Bag {
    func remove() -> Element {
        return elements.removeFirst()
    }
}
```

- Les types génériques peuvent être contraints à l'adoption d'un protocole (par exemple) :

```
func affiche<Element: Printable>(toPrint: Element)
{
    print(toPrint.toString())
}
```

- Les types associés
    - Les protocoles ne peuvent être génériques, par contre ils peuvent être définis aux types près à l'aide de typealias :

```
protocol Subscriptable {
    typealias ItemType
    subscript(idx: Int) -> ItemType { get }
}

extension Bag : Subscriptable {
    typealias ItemType = Element
    subscript(idx: Int) -> ItemType {
        return elements[elements.startIndex+idx]
    }
}

var b : Bag<Int> = Bag()
for i in 0..<10 { b.add(i) }
let at3 = b[3]
```

- Les clauses where permettent de contraindre sur les types associés :

```
protocol Set {
    typealias ItemType
    func add(item: ItemType)
}
func compare<S1: Set, S2: Set where S1.ItemType
== S2.ItemType, S1.ItemType: Number>(s1: S1,
s2: S2) -> Bool {
    return true
}
```

- Le contrôle d’accès
- Il repose sur deux concepts :
  - Le module : une unité de distribution (Framework ou Application)
  - Le fichier source : comme d’habitude…

- Les trois accès sont :
  - public (public) : accès de module à module
  - interne (internal) : accès à l'intérieur du module
  - privé (private) : accès à l'intérieur d'un source
- Attention à la définition de privé… On ne peut rien cacher dans un même source!
- L'accès par défaut est interne.

- Les règles d’usage sont :
  - pour une application auto-contenue, inutile de préciser la visibilité, internal suffit
  - pour un framework, rendre public l’API
  - pour les tests unitaires, qui nécessitent un accès particulier, il faut utiliser l’annotation @testable

- Il y a une inférence d’accès
    - pour les tuples c’est le type le plus inaccessible qui l’emporte. Attention on ne peut spécifier d’accès pour un tuple, c’est implicite!
    - pour les fonctions même règle, mais l’accès doît être explicitement spécifié
    - pour les sous-classes on ne peut qu’au mieux restreindre l’accès

- À propos des opérateurs
  - Attention! Important! Les opérateurs arithmétiques ne débordent pas, des erreurs sont générées!

```
var maVariable: UInt32 = 0xFFFFFFFF
print(maVariable)
maVariable++
Execution was interrupted, reason: EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, sub...
```

- Pour obtenir l’arithmétique usuelle on peut employer & :

```
var maVariable: UInt32 = 0xFFFFFFFF
print(maVariable)
maVariable = maVariable &+ 1
```

- Il est possible de surcharger les opérateurs

```
struct Vector {
    let x : Float
    let y : Float
}

let vec1 = Vector(x: 5.0, y: 6.0)
let vec2 = Vector(x: 5.0, y: 6.0)

func +(v1: Vector, v2: Vector) -> Vector {
    return Vector(x: v1.x+v2.x, y: v1.y+v2.y)
}

let vec3 = vec1+vec2
```

- Les opérateurs préfixes et suffixes doivent utiliser le marqueur prefix ou suffix :

```
prefix func -(v: Vector) -> Vector {
    return Vector(x: -v.x, y: -v.y)
}

let vec4 = -vec3
```

- On peut surcharger les opérateurs composés +=, -=, …
- On peut surcharger (et on doit parfois) les opérateurs d’équivalence ==, !=

- On peut définir ses propres opérateurs!

```
infix operator +-+ {}

func +-+(v1: Vector, v2: Vector) -> Vector {
    return v1
}

let vec5 = vec3 +-+ vec3
```

- La grammaire des opérateurs autorisée est définie (se reporter au guide de programmation Swift)

```
prefix operator © {}

prefix func ©(s: String) -> String {
    return "©ParisDiderot \(s)"
}

let v = ©"Bonjour!"
```

- Pour les nouveaux opérateurs infixes on peut fixer la précédence et l’associativité