* Une variable a trois informations :
  + Une valeur
  + Un nom
  + Un type
* Il y a **3  façons** de déclarer une variable
* var text: *String*
* // Le type est annoté ce qui est nécessaire car la variable n'est pas initialisé avec une valeur
* var text = "Je suis un texte"
* // Le type est déduit à partir de la valeur de la variable
* var text: *String* = "Je suis un texte"
* // Le type est annoté, ce n'est pas nécessaire car il peut être déduit. Mais ce n'est pas interdit.
* Vous connaissez désormais 4 types :
  + 3 types numériques :
    - Int  pour les nombres entiers
    - Float  pour les nombres décimaux
    - Double pour les nombres décimaux avec 2 fois plus de précisions.
  + 1 type pour les chaînes de caractères :  String
    - On peut concaténer les chaînes avec l'opérateur  +
    - On peut insérer une variable d'un autre type avec  \(nomDeMaVariable) .
* Swift est un langage strict. On ne peut pas faire n'importe quoi ! Notamment :
  + Une variable**a toujours un type**
  + Une variable **ne peut pas changer de type**
* Les boucles "for" permettent de répéter une instruction un certain nombre de fois. On les crée avec la syntaxe suivante :
* for i in 0...10 {
* // Les instructions sont répétées ici
* }
* On peut omettre la variable  i ci dessus en inscrivant à la place le caractère  \_ .
* Il existe 2 types d'intervalles (ou range en anglais) :
  + Les intervalles fermés :  0...10 , ils contiennent les deux extrémités.
  + Les intervalles semi-ouvert :  0..<10 , ils ne contiennent pas l'extrémité supérieure.
* On utilise les intervalles pour construire les boucles for.
* On peut ajouter, soustraire, multiplier par, ou diviser par un nombre une variable avec les opérateurs suivant :
  + +=
  + -=
  + \*=
  + /=
* Nous avons découvert un nouveau type :  Bool  pour booléen qui ne peut prendre que 2 valeurs :  true ou  false .
* Ce type a plusieurs opérateurs :
  + Pour créer des booléens :

|  |  |
| --- | --- |
| == | égalité |
| != | différence |
| > | Strictement supérieur |
| >= | Supérieur ou égal |
| < | Strictement inférieur |
| <= | Inférieur ou égal |

* + Pour combiner ou modifier des booléens :

|  |  |
| --- | --- |
| && | ET logique |
| || | OU logique |
| ! | Négation |

* La déclaration if / else if / else  permet de gérer différentes conditions dans un programme. Sa syntaxe est la suivante :
* if expressionLogique1 {
* // instructions
* } else if expressionLogique2 {
* // instructions
* } else {
* // instructions
* }
* Nous avons découvert la boucle while qui permet de répéter des instructions tant qu'une condition est vraie
* while condition {
* // Instructions
* }
* Nous avons découvert l'opérateur **modulo** noté  %
  + Il permet d'obtenir le reste d'une division euclidienne
  + Par définition, il n'est applicable que sur des nombres de type  Int
* Nous connaissons désormais 3 **structures de contrôles** sur les 4 existantes en Swift :
  + Les boucles  for  et  while
  + Les instructions  if / else if / else
* Un programme, c'est comme une rivière !

var notesDeLaClasse = [18, 4, 14, 15, 17, 15, 9, 20, 19, 11, 12, 14]

// On ajoute la note de Fred

notesDeLaClasse.append(13)

// On calcule la somme des notes

var somme = 0

for note in notesDeLaClasse {

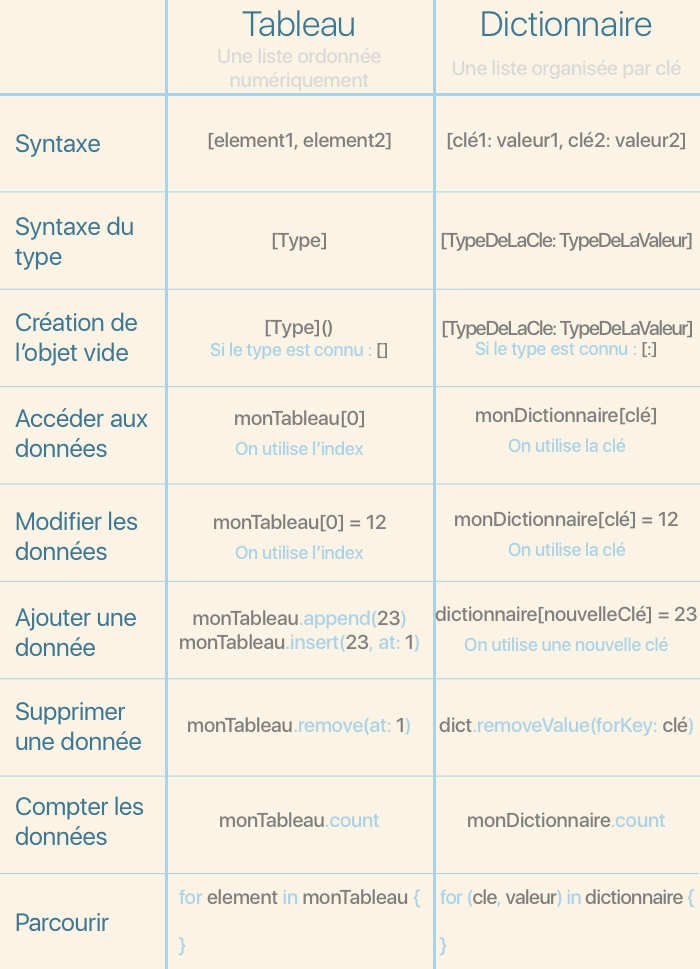
somme += note

}

// On calcule la moyenne

var moyenne = somme / notesDeLaClasse.count

* Swift propose trois types de **collections** :
  + Les tableaux
  + Les dictionnaires
  + Les ensembles
* Les tableaux sont des**listes ordonnées numériquement**
* Un tableau ne peut contenir que des valeurs d'un **type unique** et ne peut pas en changer. Le type se notant entre crochets :  [Int] . Il peut être déduit à partir des valeurs du tableau.
* Pour déclarer un tableau vide, on utilise :
  + [] si le type est connu
  + [Int]() le type suivi de parenthèses sinon
* Pour accéder ou modifier le contenu d'un tableau, on utilise l'**index entre crochets** :
* var fleurs = ["Rose", "Paquerette", "Marguerite"]
* fleurs[0] = "Iris"
* Pour ajouter ou supprimer des éléments, on utilise les fonctions :
  + append pour ajouter un élément à la fin du tableau
  + insert pour ajouter un élément à un index donné
  + remove pour supprimer un élément à un index donné
* La propriété  count permet d'obtenir la taille d'un tableau
* On peut parcourir le contenu d'un tableau à l'aide d'une boucle for
* Les variables évoluent selon un cycle de vie en 4 étapes :
  + Déclaration : La variable a un **nom** et un **type** et est **constante ou non**
  + Initialisation : La variable a une **valeur**
  + Modification : La **valeur** de la variable est modifiée
  + Suppression : La variable n'existe plus
* On peut initialiser un type en utilisant le nom du type suivi de parenthèses :  Int()



* Un optionnel est un type qui permet à une variable d’être utilisée sans valeur.
* Dans le cas où l’optionnel ne contient pas de valeur, on dit que sa valeur est  nil .
* Il faut se représenter un optionnel comme un paquet cadeau qui contient soit :
  + Une valeur du type indiqué
  + Rien
* Soit la déclaration suivante :
* var i: *Int*?

 i  **n’est pas du type entier mais du type optionnel**. Mais si  i  contient une valeur, ce sera un entier.

* Un optionnel doit donc être déballé, selon trois méthodes :
  + Une méthode rapide avec ! mais qui fera planter le programme si la variable ne contient pas de valeur.
  + optionnel!
  + Une méthode sécurisée qui permet seulement de vérifier si la variable contient une valeur ou non.
  + if optionnel != nil {
  + // optionnel contient à coup sur une valeur mais n'est pas encore déballé
  + }
  + Une méthode sécurisée qui permet d’accéder à la variable par la création d’une nouvelle variable.
  + if let déballé = optionnel {
  + // J’utilise déballé normalement ici
  + }
* Lorsqu’on accède aux valeurs d’un dictionnaire avec une clé, le dictionnaire renvoie un optionnel et non la valeur directement.
* Une fonction est **une brique d'instructions que l'on regroupe derrière un nom.**
* On déclare une fonction avec le mot-clé  func  comme ceci :
* func nomDeLaFonction() {
* // Implémentation
* }
* Une fonction peut avoir une valeur de retour. Pour cela, il faut préciser son type dans la déclaration comme ceci :
* func nomDeMaFonction() -> Type {
* // Implementation
* return valeur
* }
* Une variable n'existe que dans le contexte dans lequel (les accolades à l'intérieur desquelles) elle a été définie.

func somme(a: *Int*, b: *Int*, c: *Int*) -> *Int* {

return a + b + c

}

somme(a: 2, b: 1, c: 3) // 6

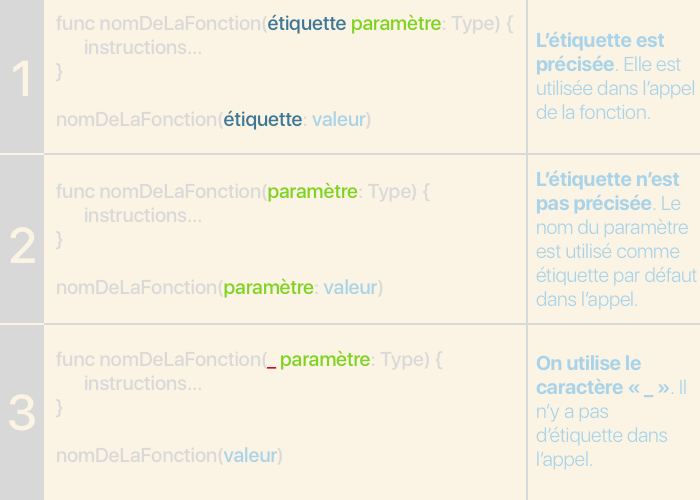
func allerAuCinema(avec nom: *String*) {

print("Je vais au cinéma avec " + nom)

}

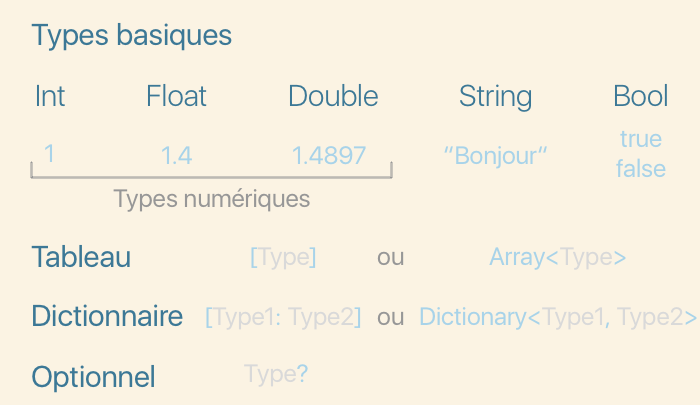
allerAuCinema(avec: "Bob") // affiche "Je vais au cinéma avec Bob"

J'ai rajouté dans la définition le motavec . C'est l'**étiquette**du paramètre nom. Du coup, dans l'appel de la fonction, on va pouvoir utiliser uniquement l'étiquette avec. Cela rend l'appel plus clair sans toucher au nom du paramètre.



* Les fonctions peuvent accepter des paramètres, qui permettent de configurer la fonction.
* Un paramètre, c'est comme une variable qui n'existe qu'à l'intérieur de la fonction. Comme toute variable, elle a un nom et un type définis comme ceci :  func nomDeLaFonction(param1: Type1, param2: Type2) {
* // Ici je peux utiliser param1 et param2 comme des variables
* }
* La valeur d'un paramètre n'est pas définie lors de la définition de la fonction, mais lors de son appel :  nomDeLaFonction(param1: valeur1, param2: valeur2)
* Les paramètres peuvent avoir des étiquettes qui permettent un appel plus lisible de la fonction. Je vous redonne mon schéma pour le résumé :
* Une fonction qui a un paramètre d'un certain type ne peut pas être appelée avec une valeur d'un type différent

Une fonction peut avoir 0, 1 ou plusieurs paramètres



<https://github.com/raywenderlich/swift-style-guide>

LES CLASSES

* Pour déclarer une classe avec Swift, on utilise le mot-clé class comme ceci :
* class NomDeLaClasse {
* // Ici viendra le contenu de la classe
* }
* Une classe s'utilise exactement comme un type. Lorsque l'on crée une classe, on crée un type.
* Une classe représente un concept. On appelle les représentations concrètes de cette classe des instances. Pour créer une instance, on utilise le nom de la classe suivi de parenthèse :
* var instance = MaClasse()
* [#](https://openclassrooms.com/courses/approfondissez-swift-avec-la-programmation-orientee-objet/creez-votre-premiere-classe#/id/r-4497205)Pour créer une propriété, on utilise simplement la syntaxe de la déclaration d'une variable à l'intérieur de la définition de la classe.
* Pour accéder à une propriété ou la modifier, on utilise le point . suivi du nom de la propriété.
* Si une propriété est constante, je ne peux JAMAIS la modifier, elle garde sa valeur par défaut.
* Si une instance est constante, je peux modifier la valeur de ses propriétés, mais je ne peux pas lui assigner un autre objet.

// Exerice 1

class RoadSection {

init() {

canvas.createRoadSection()

}

}

// Exercice 2

class Road {

var sections = [RoadSection]()

init(length: *Int*) {

for \_ in 0..<length {

self.sections.append(RoadSection())

}

}

}

var road = Road(length: 20) // Vous devriez voir une route de 20 sections se dessiner sur le canevas.

**En résumé**

* Pour utiliser des propriétés sans valeur par défaut, on peut :
  + leur donner un type optionnel : la propriété pourra ne pas avoir de valeur
  + utiliser un initialiseur
* Un initialiseur est une fonction qui est appelée lors de la création d'une instance. Pour créer un initialiseur, on utilise le mot-clé init. Comme toute fonction, un initialiseur peut avoir des paramètres.
* Le mot-clé self fait référence à la classe dans laquelle il est appelé. Il s'utilise comme une instance de la classe.
* Un objet, c'est un petit paquet de variables (les **propriétés**) et de fonctions (les **méthodes**).
* Les méthodes sont des fonctions définies à l'intérieur de la classe.
* Pour appeler une méthode, on utilise le . suivi du nom de la méthode.
* Les propriétés et méthodes de classe se déclarent avec le mot-clé static.
* Par opposition à une propriété ou méthode d'instance, une propriété ou méthode de classe s'utilise à partir du nom de la classe.
* Les propriétés de classe sont particulièrement utiles pour définir des constantes pour la classe.
* Les méthodes de classe sont particulièrement utiles pour définir des initialisations particulières ou pour gérer plusieurs instances.

**L HERITAGE / En résumé**

* Une classe (dite *fille*) peut hériter d'une autre classe (dite *mère*). Dans ce cas, toutes les propriétés et méthodes de la classe mère sont disponibles dans la classe fille sans que la classe fille ait besoin de le redéfinir.
* L'héritage permet d'organiser son code grâce à un arbre d'héritage et d'éviter ainsi des répétitions.
* Pour qu'une classe hérite d'une autre, on utilise les : suivi du nom de la classe mère :
* class Fille: Mere {
* }
* Les énumérations, comme les classes et les structures, sont des **modèles de données nommés**.
* Les énumérations permettent d'énumérer des cas.
* La syntaxe de création d'une énumération est la suivante :

enum NomDeLEnumeration {

case nomDuCas1

case nomDuCas2

case nomDuCas3

}

* Pour utiliser une énumération, on utilise le nom de l'énumération suivi de . puis du nom du cas : NomDeLEnumeration.nomDuCas1.
* On utilise très souvent les énumérations avec les switch.
* **Les optionnels sont des énumérations**.
* Les énumérations peuvent contenir des valeurs, appelées **valeurs brutes**. Ces valeurs doivent être du même type pour tous les cas d'une énumération. On les rajoute comme ceci :

enum NomDeLEnumeration: Type {

case nomDuCas1 = valeurBrute1

case nomDuCas2 = valeurBrute2

case nomDuCas3 = valeurBrute3

}

La valeur brute est contenue dans la propriété rawValue.

Modifier ou changer les caractéristiques de la classe fille par rapport à la classe mère

Autrement dit l'override permet à une classe fille de **compléter l'implémentation d'une méthode la classe mère** ou de la **réécrire complètement**. Dans les deux cas, comment fait-on ?

On utilise tout simplement le mot-clé override. Regardons ça avec un exemple. Prenons la classe Animal suivante :

class Animal {

func saluer() {

print("Bonjour")

}

func seDécrire() {

print("Je suis un animal.")

}

}

Maintenant prenons une classe fille Chien :

class Chien: Animal {

}

Pour redéfinir les méthodes, il me faut rajouter le mot-clé override avant ma fonction :

class Chien: Animal {

override func saluer() {

print("Wouf !")

}

override func seDécrire() {

super.seDécrire()

print("Et pas n'importe lequel : un chien !")

}

}

Voyons un peu ce que j'ai fait ici. J'ai modifié la méthode saluer. Désormais elle n'affiche plus "Bonjour", mais "Wouf !". La méthode a été **entièrement modifiée**, on ne réutilise rien de la méthode originale. En revanche, dans la méthode seDécrire, j'appelle la méthode seDécrire de la classe Animal en utilisant le mot clé super comme pour l'initialisation de tout à l'heure. Ensuite, je **complète** la méthode en rajoutant "Et pas n'importe lequel : un chien !". Du coup, si j'écris :

let unChien = Chien()

unChien.seDécrire()

La console affiche :

Je suis un animal.

Et pas n'importe lequel : un chien !

### En résumé

* Le mot-clé super permet d'accéder dans une classe fille à l'implémentation d'une méthode ou d'une initialisation de la classe mère.
* Le mot-clé override permet de modifier dans une classe fille une méthode définie dans une classe mère soit en la complétant soit en la redéfinissant complètement.

**Vérifier un type**

Pour cela, nous allons découvrir ensemble le mot clé is. Le mot clé is permet d'inspecter le type d'un objet. Avec cet outil, essayons de compter le nombre de chansons dans la librairie :

var nombreDeChansons = 0

for media in librairie {

if media is Chanson { // Ici on contrôle que le media est bien une chanson

nombreDeChansons += 1

}

}

print("La librairie contient \(nombreDeChansons) chansons.")

Dans la boucle for, je ne sais pas si la variable media est du type Chanson ou du type Film. La déclaration if media is Chanson qui se traduit littéralement "si le média est une chanson" nous permet de vérifier le type du média. Si c'est le cas, on incrémente notre compteur.

Le mot-clé is s'utilise donc avec la syntaxe variable is Type et renvoie un booléen.

**Accéder à un sous-type**

En reprenant, notre exemple de librairie, j'aimerais maintenant afficher tous les chanteurs de toutes les chansons de la librairie. Essayons la même stratégie :

for media in librairie {

if media is Chanson {

print(media.chanteur) // ERREUR

}

}

On a une erreur, car même si on a contrôlé le type de media, Swift considère toujours que media est juste du type Media et pas du type Chanson. Et donc la propriété chanteur n'existe pas. C'est toute la rigueur de Swift, une même variable ne changera JAMAIS de type même si on prend beaucoup de précautions.

Il va donc falloir créer une nouvelle variable et on va faire cela en utilisant l'opérateur as comme ceci :

for media in librairie {

if let chanson = media as? Chanson {

print(chanson.chanteur)

}

}

Voyons ce qu'il se passe en détail :

* Si la variable media est du type Chanson :
  + Une nouvelle variable est créée : la variable chanson de type Chanson et qui contient les mêmes données que la variable media.
  + On effectue les instructions à l'intérieur des accolades. La variable chanson y est disponible.
* Si la variable media n'est pas du type Chanson :
  + Aucune variable n'est créée.
  + On ignore les instructions entre accolades.

Il nous reste une question à voir, celle du point d'interrogation. Le point d'interrogation après le as signifie que cette opération peut échouer. En effet, la variable media peut ne pas être du type chanson.

On peut également utiliser le point d'exclamation après leassi on est **absolument certain** du type de notre variable. Par exemple :

(librairie[1] as! Chanson).chanteur

Entre les parenthèses, on force l'évaluation de librairie[1] à Chanson, car on sait que c'est une chanson. Toute l'expression entre parenthèses est donc considérée comme une variable de type Chanson dont on peut obtenir du coup la propriété chanteur.

Cela ne vous fait pas penser à quelque chose ? C'est très proche du déballage des optionnels :

// Vérification que l'optionnel n'est pas nul

if optionnel != nil {

// Faire quelque chose

}

// Déclaration optionnelle

if let déballé = optionnel {

print(déballé)

}

// Déballage forcé

let déballé = optionnel!

C'est à peu près la même logique, du coup les syntaxes sont similaires !

**Les types Any et AnyObject**

**Any**

Il y a un type un petit peu particulier que nous allons découvrir maintenant : le type Any. Une variable de type Any peut accepter une valeur de n'importe quel type :

let a: Any = 12 // Ça marche

let b: Any = "Blob" // Ça marche aussi

let c: Any = true // Ça marche encore

let d: Any = Bus() // Ça marche toujours !

Encore plus fort, une variable de type Any peut changer de valeur avec des valeurs de types différents.

var a: Any = 12

a = "Blob"

a = true

a = Bus()

Non, je ne vous ai pas menti. Une variable ne change JAMAIS de type. Ici la variable a garde toujours le même type, seulement c'est le type Any. 

**AnyObject**

Le type Any a un petit frère que vous risquez de rencontrer : le type AnyObject. Quelle est la différence ? Le type AnyObject fonctionne uniquement avec les instances d'une classe. Cela exclut notamment tous les types basiques qui sont des structures en Swift (et non des classes) comme Int, Float, Double, Bool, String. En résumé :

var a: AnyObject = 12 // Renvoie une erreur

var a: Any = 12 // OK

var b: AnyObject = Bus() // OK

var b: Any = Bus() // OK

**Usages**

OK, mais à quoi ça sert ?

Il y a deux usages principaux.

**Collections de types mixés**

Le premier usage courant, c'est de pouvoir créer des **collections de types mixés** par exemple comme ceci :

var tableauMixé: [Any] = [12, "Blob", true, Bus()]

Dans ce tableau, je peux utiliser des valeurs de différents types grâce au type Any.

**Objet de type inconnu**

Le deuxième cas courant d'utilisation, c'est quand **on ne sait pas le type d'un objet** que l'on va utiliser.

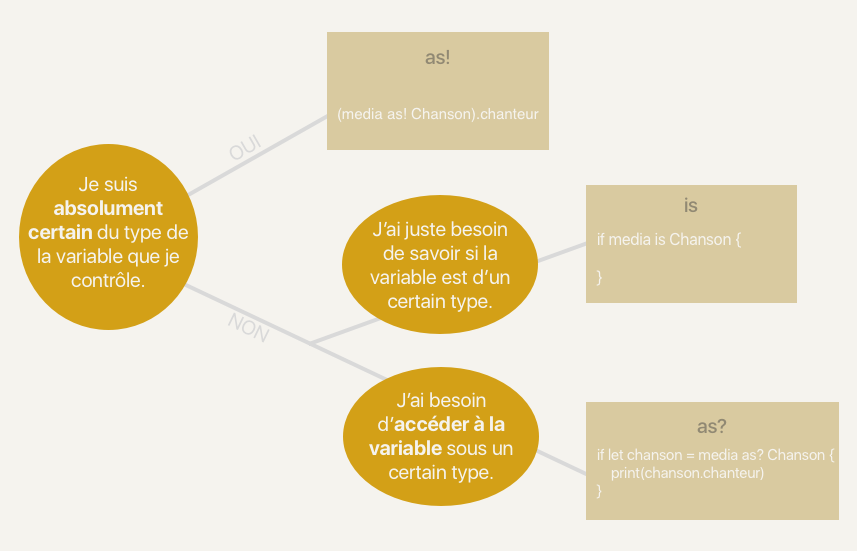
Prenons l'exemple d'une requête réseau. Je souhaite récupérer l'âge d'un utilisateur. Le serveur peut me répondre de deux façons différentes :

* Tout se passe bien, je reçois : ["age": 22]. La réponse est donc du type : [String: Int].
* Il y a une erreur, je reçois : ["erreur": "Le serveur a planté !"]. La réponse est cette fois du type [String: String]. Donc dans cet exemple, je peux recevoir une variable de deux types différents. Pour pouvoir gérer ces deux types, je vais déclarer que ma variable est du type : [String: Any].

**Vérification du type**

La raison pour laquelle je vous parle des types Any et AnyObject maintenant, c'est qu'avec des variables de ce type, on va souvent avoir besoin de vérifier le type qu'elles ont réellement avant de pouvoir les utiliser. Les types Any et AnyObject sont donc un des cas d'usage très courant de la vérification des types.

**En résumé**

* Les deux cas d'usage les plus courants de la vérification des types sont : l'héritage et les types Any et AnyObject.
* Pour vérifier un type, on a trois méthodes : is, as? et as!.
* Pour choisir la méthode à utiliser, vous pouvez vous référer à ce schéma :

Les types Any et AnyObject permettent aux variables d'accepter des valeurs de n'importe quel type (uniquement les instances de classe pour le type AnyObject). On les utilise lorsqu'on veut faire des collections de types mixés ou qu'on ne connaît pas le type d'une variable que l'on va utiliser.