KOTLIN (survol)

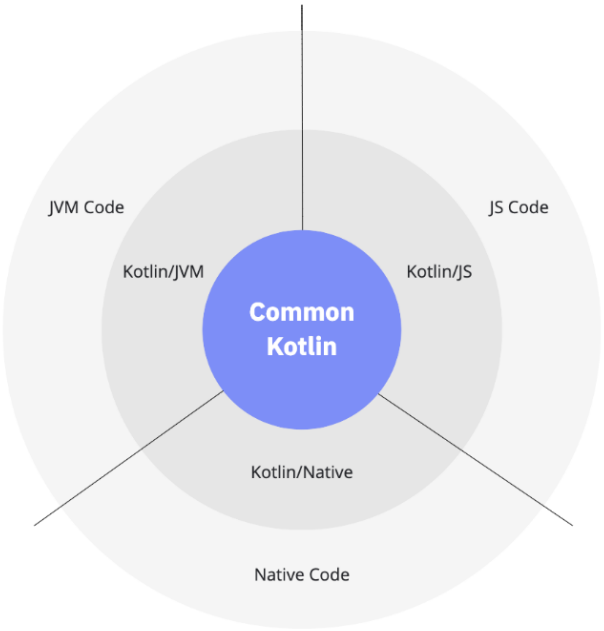
Kotlin Multiplateforme est un langage de programmation orienté objet et fonctionnel, avec un typage statique qui permet de compiler pour la machine virtuelle Java, JavaScript, et vers plusieurs plateformes en natif (grâce à LLVM - Low Level virtuel Machine- initié par l'université » de l'Illinois, maintenant sous licence Apache)

Son développement provient principalement d'une équipe de programmeurs chez JetBrains . C'est le 19 juillet 2011, lors du **JVM - Language Summit**, que **JetBrains** a présenté **Kotlin**, un nouveau langage de programmation statiquement typé pour la JVM. (Basée à Saint-Pétersbourg en Russie -son nom vient de l'île de Kotline, près de St. Pétersbourg) l'éditeur **d'IntelliJ IDEA**, l'environnement de développement intégré pour Java et sur lequel est basé **Android Studio**, l'EDI officiel pour développer les applications Android.

Google annonce pendant la conférence Google I/O 2017 que Kotlin devient le second langage de programmation officiellement pris en charge par Android après Java. Le 8 mai 2019, toujours lors de la conférence Google I/O, Kotlin devient officiellement le langage de programmation voulu et recommandé par le géant américain Google pour le développement des applications Android à partir du concept KMM.

KMM _ Kaolin Multiplateforme Mobile

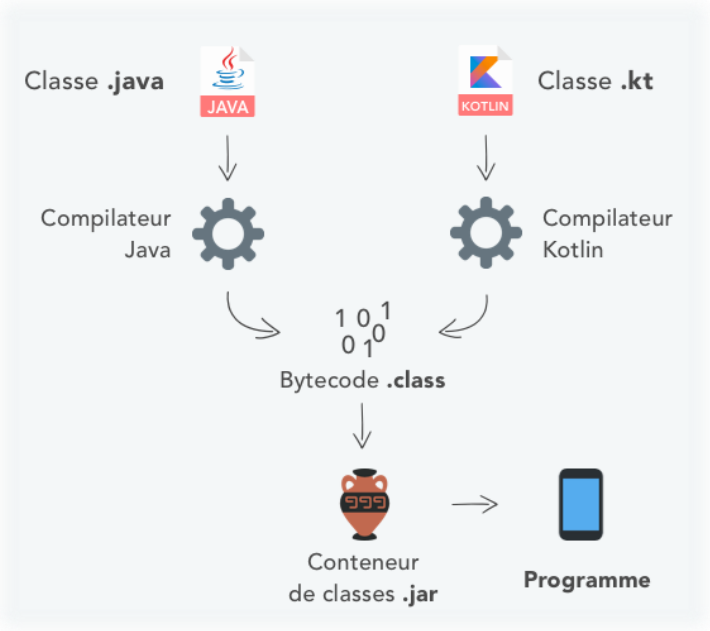
L'objectif de la technologie **Kotlin Multiplatform Mobile (*KMM*)** est d'unifier le développement d'applications avec une logique commune pour les plates-formes Android et iOS. Pour rendre cela possible, KMM utilise une structure spécifique au mobile des projets Kotlin Multiplatfor



tout projet Kotlin Multiplateforme fonctionne de la façon suivante : une partie commune et une partie spécifique :

- le code écrit en Common Kotlin contient les langages, librairies et les outils fonctionnant sur toutes les plateformes : il s'agit de la partie commune
- le code spécifique utilise des versions spécifiques de Kotlin : (Kotlin/JVM, Kotlin/Native, Kotlin/JS)

Kotlin et Java sont **100 % interopérables**, c'est-à-dire que vous pourrez, sur un même projet, écrire du code source en Java **ET** en Kotlin. Vous pourrez appeler en Java des méthodes créées en Kotlin, et inversement. Le code compilé (le bytecode) sera exactement le même.



1 - Bases du langage

1-1 Type des variables :

Kotlin est un langage typé statiquement, tout comme Java. Cela signifie simplement que le type de chaque variable est déjà connu au moment de la compilation.

- Type entier : Int, short, long, Byte
- Type flottant : Float (ajout f/F), Double
- Constances littérales : décimal→123 , Hexadécimal →0x0F, binaire →0b00001011
- Textes : String
- Exemple textes : brut → '' a b c'', échappée → '' hello\n''
- Tableaux : Array classe → création : arrayOf()

L'une des différences entre Kotlin et Java, en ce qui concerne les types, est que Kotlin prend en charge *l'inférence de type*. L'inférence de type est la possibilité de déterminer automatiquement le type d'une variable à partir du contexte.

Il y a trois *mots-clés* différents à utiliser pour déclarer des variables dans Kotlin :

- **Var**– Pour déclarer des variables *modifiables* qui peuvent changer de valeur. (muable)
- **Val** – Pour déclarer *des variables en lecture seule*.(immuable)

- **Const**– Pour déclarer des *constantes*.**cons val**
- Une variable peut être défini ultérieurement : **lateinit**
- On peut cependant définir un type pour les variables val : nom val :string = pomme.

Remarques : - une variable peut avoir une valeur NULL : on met un ? acores la variable
Var quantité : ? = stock
Pour autoriser les valeurs nulles, vous pouvez déclarer une variable en tant que chaîne **nullable** en écrivant **String?**

```
Var b String ? =''abc''     //peut être défini comme null
B = null                    // ok
Println ( b)
```

- On ne met pas de ; après une instruction

val name = "Phil'	équivalent ou égal	val name:: String = "Phil"
val age=27		val age: Int = 27
val isdevelopper = true		val isdevelopper :boolean = true

- Variable locale à l'intérieur d'une variable : utilisation de **$**
  - val name = "Phil"
  - print ("Hello $name")
  -
  - // Output : Hello Phil

1-2 - déclaration des fonctions (méthode en java)

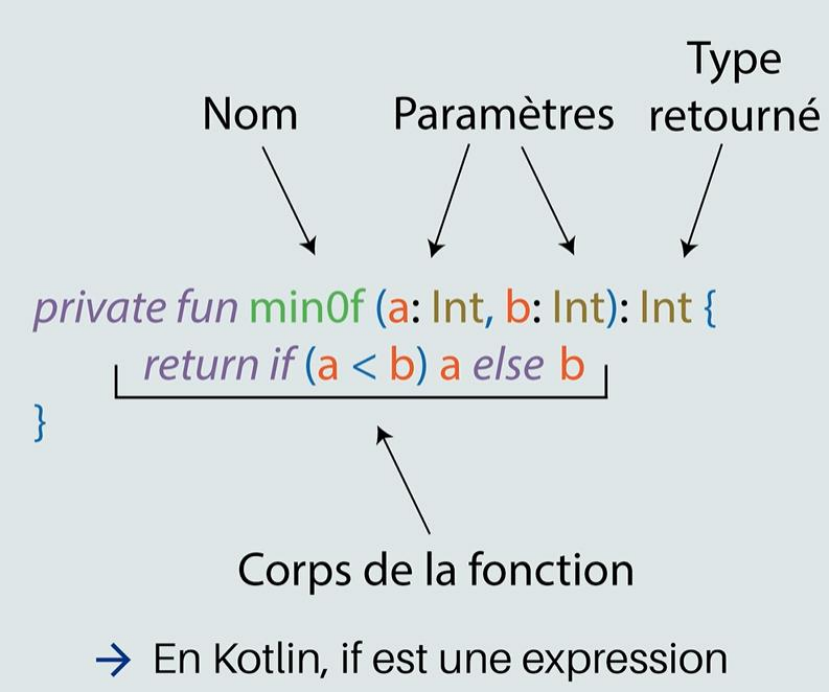
La déclaration de fonctions dans Kotlin commence par un mot-clé FUN , suivi du nom de la fonction

```
Fun salutation () {
        println("bonjour")
}
```

La fonction n'a pas de paramètres car les parenthèses sont vides ,

Paramètre d'une fonction

Ensuite, entre parenthèses, nous avons les paramètres (a, b). Et le type de retour *Int* vient après que les paramètres soient séparés par un deux-points.



En Kotlin IF est une expression et non une instruction (if peut retourner une valeur)

Si la fonction ne retourne aucune valeur, son type de retour est **Unit**.
Quand une fonction retourne une seule expression, les accolade peuvent être omit.

Voir l'exemple suivant

```
Fun somme (a : Int ,b : Int) Int=a+b
```

Dans l'exemple précédent l'opérateur **=** est utiliser pour affecter la valeur de retour de l'expression à la fonction.

1-3 Notion de classe

Une classe est un modèle de définition des objets ayant des caractéristiques identiques (même propriétés et opérations). A partir d'une classe vous pouvez créer plusieurs d'objets et chaque objet sera une instance de cette classe.

Dans Kotlin, vous pouvez déclarer une classe avec le mot clé "**class**".

Superclasse

Toutes les classes de Kotlin ont une superclasse commune, , qui est la superclasse par défaut pour une classe sans supertypes déclarés :**Any**

Any a trois méthodes : et . Ainsi, ces méthodes sont définies pour toutes les classes Kotlin. : `equals(),hashCode(),toString()`

Création d'une classe :

```
1       fun main () {
2               val personne = Personne()
3       }
4
```

```
5       class Personne
```

1-3 -1 Constructeur d'une classe : Propriétés d'une classe

Le constructeur est une méthode spéciale de classe utilisé pour initialiser les propriétés d'un objet de la classe lors de la création de l'objet. Le constructeur est appelé durant la création de la classe. Si vous ne déclarez pas un constructeur, le compilateur crée un constructeur par défaut pour votre classe qui servira à instancier la classe.

Dans kotlin, il existe deux types de constructeur : le constructeur primaire et le constructeur secondaire.

Vous pouvez déclarer des propriétés dans un constructeur primaire et les initialiser directement dans le corps de la classe (entre parenthèses): son nom , puis on type

```
1       class User (
2          var email : String
3          var password : String
4         var age : Int
5       )
```

Ou en forme compact

```
1       class User (var email : String, var password : String, var age : Int)
```

Constructeur de la classe /Constructeur primaire

On crée une instance

```
1       fun  main() {
2       val utilisateur =User (vt@gmail.com, azer12, 23)
3       class User (var email : String, var password : String, var age : Int)
4       }
```

1-3-2- Visibilité

En Kotling, la visibilité par défaut est " public "

Modificateurs de visibilité

- **Private** : visible uniquement dans la classe ou il est déclaré
- **Protected** : dans la classe ou il est déclaré ET les sous classes (héritage)

- **Internal** : visible pour tous ceux du même module (compilé ensemble)
- **Public** : visible par partout et par tout le monde

La déclaration complète d'une classe dans kotlin contient trois parties

Partie 1: Le mot clé **class** suivi du nom de la classe
Partie 2: **L'entête de la classe** constituée d'un constructeur primaire, de la liste des super classes si la classe hérite d'autres classes etc.
Partie 3: **Le corps de la classe** constitué des propriétés de la classe, des méthodes membres de la classe, des constructeurs secondaires, du bloc d'initialisation etc.

Le corps de la classe est encapsulé par des accolades. L'entête et la corps de la classe sont optionnels. Donc si la classe ne possède pas de corps, les accolades peuvent être omit comme suit.

```
Class Personne
```

1-3-3 - Initiation d'une classe : bloc d'initiation

Le constructeur primaire ne doit contenir aucun code. Le code d'initialisation de la classe peut est défini dans le bloc d'initialisation qui est préfixé avec le mot clé init .

```
1   class Personne constructeur (var nom : String, var prenom : String) {
2   init {
3   nom=nom.toUpper()
4   }  }
```

Le bloc d'initialisation est utilisé pour transformer le nom en majuscule

Le bloc d'initialisation peut aussi être utilisé pour initialiser les propriétés membres de la classe comme suit

```
1   fun main (){
2    val personne=Personne (''Dupont'',''pierre'')
3   }
4   class Personne (var nom :String, var prenom :String){
5   val datenaissance : Int
6   val email : String
7   init{
8           datenaissance =2020
9           email= vt@gmail.com
10          println ('' datenaissance est $datenaissance'')
11          println(''L'email est $email '')
12  }
```

1-3-4- constructeur auxiliaires

Si nous voulons d'autres constructeurs, il faut les déclarer à l'intérieur de la classe et les baser sur le constructeur principal. C'est donc le mot-clef ''**constructor''** qui permet de déclarer un constructeur auxiliaire.

```
1   class Personne (val nom: String, var age: Int) {

2       constructor(nom: String) : this(nom, 0) {
            // Nous pouvons même effectuer d'autres tâches d'initialisation ici
3           println("Dans un constructeur auxiliaire")
4       }
5   }
```

Une classe peut aussi avoir plusieurs constructeurs secondaires comme dans l'exemple suivant

```
1   class Personne (var nom :String,var prenom : String ){
2   constructor (nom :String, prenom :String, email :String) :this 'nom,prenom'){
3     …..}
4   constructor (nom : String, prenom : String, email : String,  age :Int) :
5           this (nom, prenom) {
5   …..}
6   }
```

1 - 4 Héritage

1 - 4 - 1 - classe de base

Classe dérivée	Classe qui hérite de la classe de base
Init Arguments	Arguments passés au constructeur de la classe de base
Définition de fonction	Fonction dans la classe dérivée dont le code est différent de celui de la classe de base
DC-Objet	”Objet de classe dérivé” Objet du type de la classe dérivée

Dans Kotlin, les classes sont **finales par défaut,** ce qui signifie qu'elles ne peuvent pas être héritées.

Pour autoriser l'héritage sur une classe, utilisez le mot clé **open** .

```
open class Thing {
  // I can now be extended!
}
```

Exemple :

Définir la classe de base :

```
open class BaseClass {
  val x = 10
}
```

Définir la classe dérivée:

```
class DerivedClass: BaseClass() {
   fun foo() {
      println("x is equal to " + x)
   }
}
```

1-4-2 - fonctions membres

Lorsqu'une classe dérive d'une classe de base, elle hérite des fonctions membres de la classe de base.

Toute comme les classes, par défaut les fonctions membres que vous ajoutez dans une classe sont marqué **final** c'est à dire qu'elles ne peuvent pas être redéfinies dans une classe dérivée.
Tout d'abord pour permettre à une fonction d'être redéfinie(implémentée) dans une classe dérivée, vous devez la marquer **open**.
Ensuite vous devez marquer la fonction membre de la classe de base avec le modificateur **override** dans la classe dérivée sinon, le compilateur vous signalera une erreur.
Voici un exemple qui illustre comment redéfinir une fonction membre d'une classe de base dans une classe dérivée

```
1   fun main(){
2   val chien=Chien (''bouledogue '')
3   chien.afficher()
4    }
5   open class Animal (val name : String){
6   open fun afficher(){
7   println (''Mon nom d'animal est $name'')
8   ]
9   }
10  class Chien(name :String) ; Animal($name){
11  override fun afficher'){
12  }
13  ]
```

2 - Conditions et boucles

2-1 - IF

Dans Kotlin, **if** est une expression : elle renvoie une valeur. Par conséquent, il n'y a pas d'opérateur ternaire () .en Java if est une instruction

```
if condition ? then : elseif
```

2-1-1 - forme traditionnelle :

```
1   fun main ( args : Array <String>) {
2   val number= -10
3   if (number > 0) { println(''nombre positif'')
4   } else {println(''nombre négatif'')
5           }
```

2-1-2 - forme Kotlin : Expression

```
1    fun main(args ; Array<String> ) {
2   val number =-10
```

```
3       val result = if(number > 0 ) {
4               ‘’nombre positif’’
5       } else {
6               ‘’nombre positif’’
7       println (result)
8       }
```

La branche ‘’else’’ est obligatoire lors dev l’utilisation if en tant qu’expression, Les accolades sont facultatives si le corps de ifn'a qu'une seule instruction.
Par exemple,

```
1       fun main(ags : ARRAY<String> ) }
2       val number = -10
3       result) if( number >0) ‘’nombre positif’’ else ‘’ nombre negatif’’
4       println(result)
5       }
```

2--1 -2 : if …. Else … if ladder

Vous pouvez renvoyer un bloc de code parmi de nombreux blocs dans Kotlin à l'aide de l’if. Else …if.. Ladder

```
1     fun main (args : Array<String>){
2      val nombre=0
3      val result= if (nombre>0 )
4               ‘’nombre négatif’’
5     else
6               ‘’nombre négatif’’
7     else
8               ‘’zéro’’
9     println (‘’nombre est $result’’)
10    }
```

2 - 2 commutateur Switch (When)

En programmation informatique, switch (« aiguillage » en anglais), parfois aussi **select** (comme en VB) ou **inspect** ou **case** of (Pascal, Modula 2) ou Match (Rust et Python) est **une instruction qui permet d'effectuer un branchement à partir de la valeur d'une variable**
En Rotling elle se prénomme When avec une syntaxe particulière
L’expression **when** peut être : une déclaration
Une instruction

2-2-1 -When utilisé comme une déclaration : similaire à l’instruction switch de Java

```
fun main(args: Array<String>) {

print ("entrer n’importe quel grand corps :")
var lb = reader.next()

when(lb) {
    "soleil" -> println("le soleil est une Etoile")
```

```
            "lune" -> println("la lune est un satelite")
            "terre" -> println("la terre est une planete")
             else -> println("je n »en sais rien")
        }
    }
```

Sortie

Entrez n'importe quel grand corps : Soleil

Le soleil est une étoile

Entrez n'importe quel grand corps : Mars

je n'en sais rien

2-2-2- when utilisé comme expression

S'il est utilisé comme expression, la valeur de la branche dont la condition est satisfaite sera la valeur de l'expression globale. En tant qu'expression, When renvoie une valeur avec laquelle l'argument correspondant , qui peut être stocké dans une variable

```
Fun main (args : Array<String>){
      Println (''Entrer le numéro du mois'')
  Val moisannee= reader. nextint()
  Val mois = When(moisannee){
    1 →''janvier''
    2 →''fevrier''
    3 →''mars''
    4 → ''avril''
    5 →'' mai''
    6 →''juin''
    7 →''juillet''
    8 →'' aout''
    9 →''septembre''
   10→''octobre''
   11→''novembre''
   12→''decembre''
  Else →{
        Println(''ce n'est pas un mois de l'année)
    }
  }
  Pour l'impression du résultat :
  Println(mois)
```

Entrer le numéro du mois : 8

Aout

2-2-3 - Classe ''énumération''

Comme en java, on peut utiliser la classe ''enum'' pour définir un type

En java

```
1   Public enum moisannee {
2    janvier
3    fevrier
 ……
```

```
12   novembre
13   décembre
14   }
```

En Kotlin : c'est une classe

```
1    enum class moisannee (cal code int) {
2    janvier (1)
3    février (2)
…
12   novembre (11)
13   décembre (12)
14   }
```

2-2-4- Vérifiez ou non la valeur d'entrée dans la plage :

En utilisant l'opérateur **in** ou **!in** , nous pouvons vérifier la plage d'arguments transmis lors du bloc. L'opérateur 'in' dans Kotlin est utilisé pour vérifier l'existence d'une variable ou d'une propriété particulière dans une plage. Si l'argument se situe dans une plage particulière, l'opérateur in renvoie vrai et si l'argument ne se situe pas dans une plage particulière, alors **!in** renvoie vrai.

```
Fun main (args : Array<String>){
  Println (''Entrer le numéro du mois'')
  Var num =  reader.nextInt()
  When (num){
    In 1..3   →println(''c'est la saison du printemps'')
    In 4..6  → println('c'est la saison estivale'')
    In 7..8  → println('c'est la saison des pluies'')
  Else → {
      Println ( ''ce n'est pas un mois de l »année '')
  Par une intruction qui  }
 }
Println(mois)
```

Sortir:

Entrez le numéro du mois : 5

C'est la saison estivale

3- Boucle

3-1 boucle for

La boucle for au sens classique java n'existe pas :

```
For (int i=1 ;i=12 :i++){
    Expression avec $i
}
```

non en kotlin

Elle est remplacée par une fonction qui définit les intervalles

```
For ( 1 in &…12){
    Expression avec $i
}
```

oui en kotlin

En kotlin l'intervalle ''in'' ou ''lin'' couvre les mêmes besoins

En kotlin, un intervalle est toujours fermé et inclusif

```
1   for (1 in 0..3) println( i )
```

3-1-1 Forme générale :

```
1   for (item in collection) {
2       ligne de code à executer //print item
3   }
```

3-1-2 Instruction : **until**

Pour parcourir sur tous les éléments de l'intervalle sauf la limite supérieur de l'intervalle,vous devez utiliser l'opérateur **until** comme suit

```
1   for( i in 1 until 5)
2       println (''item$i'')
```

Resultat : item1 item2 item3 item4

3-1-3 Instruction de saut : **step**

Vous pouvez aussi parcourir à travers un intervalle par saut d'un valeur n en utilisant les opérateurs **in** et **step** comme suit.

```
1   for(i in 1 ..10 step2)
2       println (''item$i'')
```

Résultat : item1 item3 item5 item7 item9

3-1-4 Instruction de parcours de la valeur supérieur vers le minimum un pas : downTo,step

```
1   for( i in 10 downTo 1 step2)
2       println (''item$i)
```

Résultat : item10 item8 item6 item4 item2

3-1-5 Parcourir un tableau **: Array**

```
1   val names = arrayof (''zoé'', ''jacques'' , ''jeanne'',''pierre'')
2    for(name innames )
3   prinln(''names'')
```

Résultat : zoé jacques jeanne pierre

3-2 - boucle While /do while

While et do-while les boucles exécutent leur corps en continu tant que leur condition est satisfaite. La différence entre eux est le temps de vérification des conditions :

1. While vérifie la condition et, si elle est satisfaite, exécute le corps, puis revient à la vérification de la condition.
2. do-while exécute le corps, puis vérifie la condition. S'il est satisfait, la boucle se répète. Ainsi, le corps de do-while 'exécute au moins une fois, quelle que soit la condition

```
while ( x>0 ) {
    x--
}
Do {
     Val y= retrieveData()
} while  ( y != null )          //
```

Exemple // programme pour ecrire 5 lignes

```
Fun main (args : Array<String>) {
    Val i = 1
While ( i < 5) {
```

```
        Println(''Line $' '')
            I++
    }
    }
```

Resultat :

Line1

Line2

Line3

Line4

Line5

Si le corps de la boucle n'a qu'une seule instruction, il n'est pas nécessaire d'utiliser des accolades { }

3-3 - smart class (transcriptase intelligent)

3-3-1- vérification de type

Dans Kotlin, nous pouvons vérifier le type de certaines variables à l'aide de **l'opérateur is** au moment de l'exécution. C'est un moyen de vérifier le type d'une variable au moment de l'exécution pour séparer le flux pour différents objets.

```
fun main(args: Array<String>) {
    var name = "Praveen"
    var age = 24
    var salary = 5000.55
    val employeeDetails: List<Any> = listOf(name,age,salary)

    for(attribute in employeeDetails) {
        if (attribute is String) {
            println("Name: $attribute")
        } else if (attribute is Int) {
            println("Age: $attribute")
        } else if (attribute is Double) {
            println("Salary: $attribute")
        } else {
            println("Not an attribute")
        }
    }
}
```

Sortie:

Name: Praveen

Age: 24

Salary: 5000.55

3-3-2 - casting intelligent

En Java ou dans d'autres langages **de programmation**, il est nécessaire de lancer explicitement la variable avant d'accéder aux propriétés de cette variable, mais Kotlin effectue une diffusion **intelligente**. Le compilateur Kotlin convertit automatiquement la variable en une référence de classe particulière une fois qu'elle est passée par un opérateur conditionnel.

3 -3-3 - changement de type

AS : Vous avez une variable générique de type Any , que vous souhaitez convertir (ou "cast") en une chaîne de caractères (String).
Pour effectuer cela en Kotlin, vous allez devoir utiliser le mot-clé **as**

```
val anyObject: Any = "Hello bonjour "
val message = anyObject as String
print(message)
```

en cas d'erreur (valeur nulle), il faut gérer une exception du type **try/catch**
AS ?,en cas d'erreur (string en Int) la valeur sera ''null''

Exemple :

```
val anyObject: Any = "Hello bonjour !"        // string
val message: Int? = anyObject as? Int         // int
print(message)
```

resultat = > null

4 - gestion des exceptions

4-1-bloc try/catch

En Kotlin, une exception se gère comme en Java, grâce aux mots-clés **try et catch**.
Contrairement à Java, en Kotlin, une fonction pouvant renvoyer une exception n'a pas besoin du mot-clé **throw**s dans sa signature.
try , catch et throw sont des expressions en Kotlin et pourront donc renvoyer une valeur.

```
try
{
  // votre code
}
catch(ex : nom de l'exception
{
  // Exception handle code
}
finally
{
  // code exécuté chaque fois qu'il a une exception ou pas
}
```

Exemple :

```
try {
    val message = "Bienvenu sur Kotlin Tutoriels"
   message. toInt()
} catch (exception : NumberFormatException) {
// ... }
```

Le bloc try/catch peut être utilisé comme expression

```
val number = try {
val message = "Bienvenu sur  Kotlin Tutoriels"
message. toInt()
} catch (exception : NumberFormatException) {
// ... } return number
```

4-2 - Blocs de capture multiple

Nous pouvons utiliser plusieurs blocs *catch avec le bloc try* dans Kotlin . En particulier, cela est souvent nécessaire si nous effectuons différents types d'opérations dans le bloc *try , ce qui augmente la probabilité d'attraper plusieurs exceptions.*

```
try {
 val result = 25 / 0
  result
 } catch (exception : NumberFormatException) {
          // ...
 } catch (exception : ArithmeticException) {
         // ...
 } catch (exception : Exception) {
```

4-3 -Opérateur Elvis = ?:

L'opérateur Elvis **(?:)** est utilisé pour renvoyer la valeur not null même l'expression conditionnelle est null. Il est également utilisé pour vérifier la sécurité nulle des valeurs.

Dans certains cas, nous pouvons déclarer une variable qui peut contenir une référence nulle. Supposons qu'une variable *str* qui contient une référence null, avant d'utiliser *str* dans le programme, nous allons vérifier sa nullabilité. Si la variable str est trouvée comme n'étant pas null, sa propriété utilisera sinon une autre valeur non nulle.

```
1   var str : String? = null
2   var str2 : Chaîne ? = « Peut être déclaré chaîne nullable »
3   var len1: Int = if (str != null) str.length else -1
4   var len2:  Int = if (str2 != null) str.length else -1
```

Résultat :

Lenght of str is -1

Lenght of str2 is 30

4-4-utilisation du mot clé : throw

Nous pouvons utiliser le mot clé ***throw*** dans Kotlin pour lever une certaine exception ou une exception personnalisée.

exemple :

```
val message = "bienvenu sur Kotlin Tutoriels"
if (message.length > 10) throw IllegalArgumentException("String is invalide")
else return message.length
```

4 - 5 - classe Nothing

Nothing n'a pas d'instances.

Vous pouvez utiliser **Nothing** pour représenter "une valeur qui n'existe jamais": par exemple, si une fonction a le type de retour Nothing, cela signifie qu'elle ne revient jamais (lève toujours une exception).

Nothing est un type dans Kotlin qui **représente "une valeur qui n'existe jamais"** , cela signifie simplement "aucune valeur du tout". Nothing ne peut être utilisé comme type de retour d'une fonction qui ne retourne jamais l'exécution du code - comme, en boucle pour toujours ou toujours lève une exception.

5 - différentes variantes

5-1 - Multiplateforme Kotlin (mode alpha)

La prise en charge de la programmation multiplateforme est l'un des principaux avantages de Kotlin. Il réduit le temps passé à écrire et à maintenir le même code pour différentes plates-formes tout en conservant la flexibilité et les avantages de la programmation native.

Plateforme prise en compte :

- **Kotlin/JVM** → JAVA
- **Kotlin/JS** → **Javascript**
- **Bibliothèque Android**
- **NDK Android** hôte Linux ou macOS
- **IOS (iOS ARM 32/64-iosX64**
- WatchOS/… hôte MacOS
- TvOS /… hôte MacOs
- Linux(ARM/X64) hote Linux

5-2 - binaire natif

Pour déclarer des fichiers binaires natifs finaux tels que des exécutables ou des bibliothèques partagées, utilisez la binaries,propriété d'une cible native.

5-3 SDK

- SDK Cloudinary
- SDK GotHub ThenewBoston (android et destop)
- SDK AWS

5-4 IDE

- Intellij IDEA de Jetbrains
- Android Studi

6 - ouvrages

6-1 en français

- Les fondamentaux du langage - ENI -Ludovic Roland
- Les fondamentaux du développement d'applications Android- Upsilon-Anthony Cosson
- Développement vidéo : développement Android natif et serveur - ENI - Ludovic Roland & Anthony Cosson
- Programmer avec Kotlin - Dunod -Josh Skeed & David Greenhalg
- Développer une application Android - Eyrolles- Ubelkannt
- Développement d'applications mobiles -Ellipse - Patrick Auxerre
- Développer des applications Android - Eyrolles - Hugues Bersini
- Kotlin-guide pratique-First interactive -Ken Kousen

6_2 en anglais

- Head first Kotlin-O-Reilly- Dawn Griffiths
- Kotlin in Action -Manning publications - Dimitri Jemerov/svetlana Isakova
- The Joy of Kotlin - Manning publications-Pierre-Yves Saumon
- Programming Kotlin : create elegant-pragmatic bookshelf -venkat Subramaniam
- Learn Kotlin programming-packt publishing-Stephen Samuel
