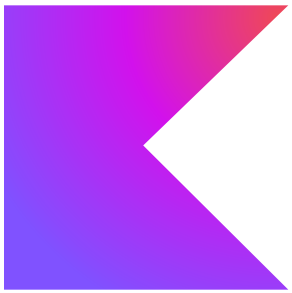

Kotlin

IUT Clermont Auvergne

Janvier 2024

Introduction à Kotlin

Pourquoi Kotlin

Google I/O 2017 : langage de premier ordre pour Android

- Beaucoup d'inférence de type, donc moins verbeux que Java
- Multi-paradigmes (procédural, fonctionnel, orienté objets)
- Pousse aux *best practices*
- S'exécute sur une JVM (et par extension sur ART)

- Langage à typage statique
- Intégré à Android Studio et IntelliJ IDEA

Pour l'auto-apprentissage :

- https://kotlinlang.org/docs/reference/
- https://try.kotlinlang.org/

Règles de base

- Code contenu dans des fichiers `.kt`
- Pas besoin de `;` en fin de ligne
- Possibilité de fonctions de premier ordre

```
fun main(args: Array<String>) {
  println("Hello Kotlin")
}
```

- Déclarations à la UML
- Conventions de nommage très similaires à Java

https://kotlinlang.org/docs/reference/coding-conventions.html

Bases du langage

Types

- Pas de types primitifs en Kotlin
- Tout est objet
- Pour les types ayant un équivalent primitif en Java
  - Le type primitif est utilisé dans la JVM
  - Sauf si contexte de généricité ou type nullable dans ce cas le type wrapper est utilisé

Types numériques

- `Double`, `Float`, `Long`, `Int`, `Short`, `Byte`
- Littéraux : comme en Java (`0b101010`, `0xB8`, `12`, `3.14f`)
- Possibilité de séparateur `1_000_000`
- Pas de conversion implicite : `toInt()`, `toFloat()`, ...

```
val theAnswer = 42;                    // OK, theAnswer de type Int
val doubleAnswer1: Double = 42;        // KO, 42 pas de type Double
val doubleAnswer2 = 42.0;              // OK
val doubleAnswer3: Double = theAnswer.toDouble(); // OK
```

- Les types entiers ont une version non-signée (`ULong`, `UInt`, `UShort`, `UByte`)

Types textuels

- `Char`, `String`
- Les chaînes de caractères sont *immuables*
- Accès au ième caractère : comme en C++ `str[i]`
- 2 types de littéraux
  - Classique (escaped string) `val str = "Hello\tTab"`
  - Sans interprétation (raw string)

```
val text = """
    \n tel quel \\
    avec saut de ligne possible
       respecte l'indentation
    """
```

  - `trimMargin()` pour alignement

Type String

- Concaténation avec opérateur `+`
  - ▷ Crée un nouvelle chaine comme résultat
- Utilisation possible de *string templates* :

```
val name = "Laurent"
val greetings = "Salut $name !"
val scream = "EH HO… ${name.toUpperCase()} !"
```

- Favoriser les *string templates* plutôt que les `+` successifs
- *string templates* autorisés dans les *raw strings*

Types classiques autres

- Booléens : `Boolean` (`true` et `false`)
- Tableaux : classe `Array<T>`
  - accès éléments : opérateur `[]` (`get`, `set` par derrière)
  - taille : propriété `size`
  - fonctions utilitaires : `arrayOf(…)`, filtres, recherche, tri, …
- `IntArray`, `DoubleArray`, etc. version optimisée pour utiliser type primitif sur JVM
- Action sans résultat : singleton `Unit` (≈ `void` de Java)
- `Nothing` : type représentant une valeur qui n'existe pas
  - p. ex. type de retour d'une fonction qui lève une exception

Les intervalles (Range et Progression)

- Opérateur `..`
- Fonctions d'extension : `until`, `downTo`, `step`

```
1..10                  // de 1 inclus jusqu'à 10 inclus
1 until 10             // de 1 inclus jusqu'à 10 exclus
4..1                   // vide
4 downto 1             //  4, 3, 2, 1
10 downto 1 step 2 // 10, 8, 6, 4, 2
```

- `step` doit être strictement positif

```
val chiffres = 0..9;
if (3 in chiffres) println("3 est un chiffre")
for (i in chiffres) print("$i, "); println("sont des chiffres")
```

Les différents type de variables (références)

- `val` : référence une valeur constante (immuable)

```
val answer: Int = 42
val age = 33;           // Type inféré
age += 2                // Erreur, pas modifiable

// Uniquement pour les variables locales
val what: String        // Type nécessaire si pas de valeur
what = "Whaaat ?"       // Initialisation différée
```

- `var` : référence une variable (peu changer de valeur)

```
var x: Int = 2;
var y = 'y';
++x;                    // OK, muable
```

- Préférez `val` à `var` quand c'est possible !
- Permet au compilateur de faire des optimisations

Les différents type de variables (références)

- `const` : constante connue à la compilation
  - soit top level, soit membre d'un `object`
  - initialisée avec une `String` ou un valeur primitive
  - pas de getter personnalisé

```
const val PI_APPROX: float = 3.14f;
```

- Comparaison de références
  - `==` comparaison structurelle (`equals()`)
  - `===` comparaison d'instances (emplacement mémoire)

Types nullables

- Exception fréquente : `NullPointerException` (NPE)
  https://en.wikipedia.org/wiki/Null_pointer#History
- Souvent due à un oubli d'initialisation
- Nullable = explicitation de valeur `null` possible
- On ajoute `?` après le nom du type

```
var maybeAnswer: Int? = null  // OK, maybeAnswer est nullable
val dummyAnswer: Int = null   // KO, dummyAnswer pas nullable
maybeAnswer = 42              // OK, peut stocker un Int
val answer: Int = maybeAnswer // KO, Int et Int? pas compatibles
```

Types nullables

- On ne peux pas utiliser *directement* l'opérateur `.` sur un nullable
- Il faut être explicite sur le fait qu'on manipule un nullable
- Deux possibilités :
  1. On vérifie avant si c'est nul

     ```
     val str: String? = …
     if (str != null) str.uppercase()
     ```

  2. On utilise à la place l'opérateur `?.` (*safe call*)

     ```
     val str: String? = …
     str?.uppercase()
     ```

Types nullables

- Si on veut une valeur par défaut pour le cas `null`
- Elvis operator : `?:`

```
val str: String? = …
val len: Int = if (str!= null) str.length else -1
// équivalent à
val len = str?.length ?: -1
```

- Forçage : `!!` (À ÉVITER !)
- Lève une NPE si l'objet est `null`
- Force le déréférencement peu importe la valeur du nullable

```
var maybeAnswer: Int? = 42
val answer: Int = maybeAnswer!! // OK maybeAnswer pas null
val str: String = null
str!!.uppercase()                // Ooops… NPE
```

Type plateforme

- Kotlin compilé en bytecode Java
- Utilisation possible des types Java
- Kotlin ne sait pas si le type est nullable ou pas
- Il est noté `Type!`
- `Type!` signifie `Type` ou `Type?`
- Fréquent sous Android

```
val date = Date()
val instant = date.toInstant()  // typé Instant!
```

- À manipuler avec précaution

Vérification de type et smart cast

- Savoir si un objet est d'un certain type : `is`

```
val myst = …
if (myst is String) {
    print(myst.uppercase())
}

if (myst !is String) { // equivalent à !(obj is String)
    print("Not a String")
}
```

- Le compilateur « se souvient » de la vérification et cast automatiquement dans la portée (smart cast)

```
// x automatiquement casté en String à droite du &&
// et dans le if
if (myst is String && myst.length > 0) {
    println(myst.uppercase())
}
```

Le transtypage (cast)

- On peut caster avec l'opérateur `as` (unsafe cast)
- Lève une exception si pas possible

```
// B dérive de A
val a: A = B()
val b: B = a as B
```

- Cast avec opérateur `as?` (safe cast)
- Le résultat est d'un type nullable

```
val a: A = …
val b: B? = a as? B
```

Contrôle du flot d'exécution

- `if` : comme en Java
- `when` : étude de cas (équivalent du `switch` … mais plus sympa)

```
when (x) {
  0, 1 -> print("peu")
  in 2..10 -> print("moyen")
  is String -> print("${x.trim()} est une String")
  else -> {
    print("rien de tout ça")
    println("on peu mettre un block")
  }
}
```

Contrôle du flot d'exécution (version *expression*)

- `when` et `if` peuvent être utilisées comme expressions
  - ▷ elles renvoient comme valeur le résultat de leur dernière instruction exécutée

```
val parite = if (x.isOdd()) "Impair" else "Pair"

println(
  when {
    x.isOdd() -> "impair"
    x.isEven() -> "pair"
    else -> "bizarre"
})
```

- Dans ce cas, les cas du `when` doivent couvrir le domaine de la variable

Boucles

- `while`, `continue`, `break` : comme en Java
- `for` : parcours sur tout ce qui fournit un itérateur (`foreach` de C#), i.e. :
  - fournit une méthode `iterator()`
  - cet itérateur a une méthode `next()` et une méthode `hasNext()` qui retourne un `Boolean` (doivent être marquées `operator`)
- Itération classique (intervalle, `String`, `Array`, …)

```
for (i in 0..9) {
  println(i)
}
for (c in "Hello") {
  println(c)
}
```

Boucles

- Itération avec indices

```
for (i in array.indices) {
  println(array[i])
}

for ((index, value) in array.withIndex()) {
  println("The element at $index is $value")
}
```

Boucles

- Possibilité de labels : `unLabel@` … `break@unLabel`

```
loop@ for (i in 1..100) {
  for (j in 1..100) {
    if (…) break@loop
  }
}
```

- À réserver pour sortir d'une imbrication de boucle

Exceptions

- Pour les exceptions : comme en Java (sauf qu'il n'existe pas de notion de *checked exception*)
- `try/catch/finally` est un expression

```
val line: String? = try {
  input.readline()
} catch (e: NumberFormatException) {
  null
}
```

- L'exécution du potentiel bloc `finally` n'impacte pas la valeur de retour

Les fonctions

Les fonctions

- Possibilité de fonction *top level* (pas forcément membre d'un objet)

```
fun sum(a: Int, b: Int): Int {
  return a + b
}
```

- Si une seule instruction : *expression body*

```
fun sum(a: Int, b: Int) : Int = a + b
```

- Avec inférence du type de retour pour les *expression body*

```
fun sum(a: Int, b: Int) = a + b
```

Les paramètres

- Paramètres avec valeur par défaut (pratique pour les Ctor)

```
fun say(text: String = "Something") = println(text)
say()                          // Affiche Something
say("Hi guys")                 // Affiche Hi guys
```

- Paramètres nommés

```
fun Box.setMargins(left: Int, top: Int, right: Int, bottom: Int) { … }

myBox.setMargins(10, 10, 20, 20)   // haut ? bas ? gauche ? droite ?
myBox.setMargins(left = 10, right = 10, top = 20, bottom = 20)
```

- On peut mixer les deux

Arguments variables

- `vararg` : nombre de paramètres variables (`Type...` de Java)

```
fun foo(vararg strings: String) { … }
foo("bar")
foo("bar", "baz", "foobar", "barbaz")
```

- Utilisation du *spread* operator (`*`) si arguments dans tableau

```
val strings = arrayOf("abc", "defg", "hijk")
foo(*strings)
```

Fonctions imbriquées

- On peut déclarer une fonction dans une autre fonction
- Encapsulation maximale

```
fun dfs(graph: Graph) {
  val visited = HashSet<Vertex>()

  fun dfs(current: Vertex) {
    if (!visited.add(current)) return
    for (v in current.neighbors)
      dfs(v)
  }

  dfs(graph.vertices[0])
}
```

Typage des fonctions

- Les fonctions sont des « symboles » comme les autres pour le compilateur
- Elles sont associées à un type :
  - `(A, B) -> C` : une fonction qui prend 2 paramètres le premier de type `A`, le second de type `B` et retourne un `C`
  - `() -> A` : pas de paramètre en entrée
  - `(A) -> Unit` : pas de valeur de retour
  - `A.(B) -> C` : fonction qui peut être appelée sur un objet de type `A`, prend un paramètre de type `B` et retourne une valeur de type `C`

Lambdas

- Les fonctions peuvent prendre en paramètres d'autres fonctions
- Exemple du `forEach` sur les collections
- Comment on lui passe une instance d'une telle variable ?
  - ▷ Grâce aux lambda

```
val square = { num: Int -> num * num } // (Int) -> Int
val more : (String, Int) -> String = { str, num -> str + num }
val printVal : Int -> Unit = { num -> println(num) }
```

- Utilisation

```
val a = arrayOf(1, 2, 3, 4, 5)
a.forEach(printVal)
// équivalent à
a.forEach({num -> println(num)})
```

Lambdas

- Pour les lambdas qui ne prennent qu'un paramètre : `it`

```
val printVal : Int -> Unit = { println(it) }
val concatInt : String.(Int) -> String = { this + it }
```

- Une fonction de type `A.(B) -> C` peut être utilisée en lieu et place de `(A, B) -> C` et vice versa
- Lorsque le dernier paramètre d'une fonction est une fonction, si on passe une lambda on peut la sortir des parenthèses
- Possibilité de passer une référence à une fonction/méthode existante grâce à l'opérateur `::`

```
val a = arrayOf(1, 2, 3, 4, 5)
a.forEach(::println)
```

Fonctions d'extension

- Possible d'ajouter des fonctions à une classe après coup
- Toutes les instances peuvent en profiter

```
fun String.reverse() = StringBuilder(this).reverse().toString()
"That's cool !".reverse()   // "! looc s'tahT"
```

Fonctions infixes

- Déclarées grâce au mot-clé `infix`
  - Fonction membre ou fonction d'extension
  - Un seul paramètre
  - Pas de `vararg` ou de paramètre par défaut

```
infix fun String.open(rights: Acces): File { … }

"/home/provot/lecture" open Access.WRITE
// équivalent à
"/home/provot/lecture".open(Access.WRITE)
```

Documentation

- KDoc + génération avec Dokka
- Comme la javadoc
- Tags : `@param`, `@return`, `@constructor`, `@receiver`, `@property`, `@throws`, `@exception`, `@sample`, `@see`, `@author`, `@since` et `@suppress`

```kotlin
/**
 * A group of *members*.
 *
 * This class has no useful logic; it's just a documentation example.
 *
 * @param T the type of a member in this group.
 * @property name the name of this group.
 * @constructor Creates an empty group.
 */
class Group<T>(val name: String) {
    /**
     * Adds a [member] to this group.
     * @return the new size of the group.
     */
    fun add(member: T): Int { ... }
}
```