

R3.04 : Qualité de développement

Rappels Kotlin

Arnaud Lanoix Brauer
Arnaud.Lanoix@univ-nantes.fr



IUT Nantes
Pôle Sciences et technologie

Nantes Université

Département informatique

Kotlin

- Langage de programmation **orienté objet et fonctionnel**
- Développé à partir de 2010 par JetBrains et de nombreux autres contributeurs (complètement open-source)
- **100 % interopérable** avec Java
 - ▶ Langage compilé : le bytecode (entre autre)
 - ▶ Machine virtuelle : la JVM – Java Virtual Machine
 - ▶ Multiplateforme
- Philosophie : *"plus concis, plus pragmatique, plus sûr que Java"*
- Langage **"fortement recommandé"** par Google pour le **développement Android** à partir de 2019
- Kotlin également compatible avec *Javascript (JS)*¹, du code natif, etc.
- <https://kotlinlang.org/>



vs.



- 1 Les bases du langage
- 2 Classes et objets
- 3 Variables == références
- 4 L'héritage
- 5 Les collections
- 6 Les exceptions

Les variables `var` ou `val`

```
val monNom = "Arnaud Lanoix"  
var monAge : Int = 42
```

En Kotlin, on manipule deux sortes de variables :

- Des variables classiques, dite muables, grâce à `var` pour "variable"
- Des variables **immuables**, grâce à `val` pour "valeur", c-à-d des variables non-modifiables, une fois initialisées (= "en lecture seulement")

Indiquer le type d'une variable n'est pas forcément nécessaire : le compilateur **déduit automatiquement** le type des variables, quand c'est possible.

Les conditions `if... else...` et `when...`

```
if (cptAbs >= 5) {  
    println("Echec($cptAbs abs)")  
}  
else if (cptAbs == 4) {  
    println("Alerte rouge($cptAbs Abs)")  
    println("* alerter tuteur *")  
}  
else if (cptAbs in 1..3)  
    println("Attention($cptAbs abs)")  
else  
    println("Pas d'absence")
```

```
var max = if (a >= b) {  
    println("$a plus grand que $b")  
    a // la dernière instruction  
    //du bloc est retournée  
}  
else if (a < b) {  
    println("$a plus petit que $b")  
    b  
}  
else b
```

```
msg = when (cptAbs) {  
    in 5.. Int.MAX_VALUE -> "Echec ($cptAbs abs)"  
    4 -> {  
        println("Alerte rouge ($cptAbs abs)")  
        "* alerter tuteur *"  
    }  
    in 1..3 -> "Attention ($cptAbs abs)"  
    else -> "Pas d'absence"  
}
```

Les boucles `while` et `for`

```
var cptRebourd = 10

println("Depart dans...")
while (cptRebourd >= 0) {
    println(cptRebourd)
    cptRebourd--
}

println("Go !!!")
```

- les boucles `while` sont à utiliser quand on ne peut pas "prévoir" le nombre d'itérations

```
println("Depart dans...")
for (cpt in 10 downTo 0) {
    println(cpt)
}
```

```
println("Depart a 10...")
for (cpt in 0 until 10 step 2) {
    println(cpt)
}
```

Les fonctions

Une fonction est définie par :

- le mot-clef `fun`
- un nom
- (éventuellement) des paramètres et leurs types
- (éventuellement) un résultat typé et renvoyé (`return`)

```
fun mult(a : Int, b : Double = 1.5, c : Double) : Double {  
    var resultat = a * b * c  
    return resultat  
}
```

- Les paramètres sont **immuables**
- Les paramètres peuvent avoir des valeurs par défaut
- A l'appel d'une fonction, on peut nommer les paramètres et modifier l'ordre d'appel

- 1 Les bases du langage
- 2 Classes et objets**
- 3 Variables == références
- 4 L'héritage
- 5 Les collections
- 6 Les exceptions

Déclarer une classe en Kotlin

```
class Chien {  
    private var nom :String = ""  
    private var age : Int = 0    // en mois  
    private var race : String = ""  
    private var poids : Double = 0.0 // en kg  
  
    fun aboyer() {  
        println("$nom dit : ouaf !!!")  
    }  
  
    fun renommer(nouveauNom : String) {  
        nom = nouveauNom  
    }  
  
    // @param distance en m  
    fun courir(distance : Int) {  
        // le chien perd 1 g / km  
        poids -= (distance / 1000.0) / 1000  
    }  
  
    fun ageEnAnnee() = age / 12.0  
}
```

- Une classe est déclarée grâce au mot-clef `class`
- Les **attributs** sont déclarés comme des variables **internes** à la classe
 - ▶ Les attributs doivent être initialisés
- Les **méthodes** sont déclarées comme des fonctions **internes** à la classe
 - ▶ On peut **consulter** ou **modifier** les valeurs des attributs via les méthodes

Constructeur en Kotlin

```
class Chien (monNom : String, race : String = "inconnue", poids : Double) {  
    private var nom : String  
    private var age : Int = 1           // en nb de mois  
    private val race : String  
    private var poids : Double // en kg  
  
    init {  
        nom = monNom  
        this.race = race  
        this.poids = if (poids > 0.0) poids else 0.1  
    }  
}
```

- les **paramètres** du constructeur sont déclarés après le nom de la classe
- On peut définir des **valeurs par défaut** pour les paramètres
- tous les attributs ne sont pas forcément présents comme paramètres
- les paramètres du constructeur servent à **initialiser** les attributs
- tous les attributs doivent être initialisés

Attribut immuable

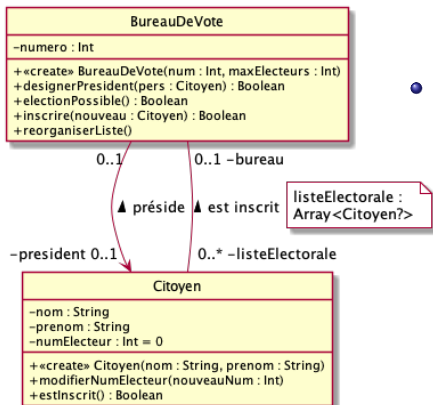
Notez qu'ici l'attribut `race` est `val` : pour un chien donné ne doit plus pouvoir être changé : il est impossible de changer sa race après sa création

Visibilités en Kotlin : exemple

```
class Chien (...) {  
    var nom :String           // A PROSCRIRE ABSOLUMENT  
    private var age : Int     // ok  
    val race : String         // possible  
    var poids : Double private set // possible  
  
    fun courir(dist : Int) {   // ok  
        poids -= poidsEnMoins(dist)  
    }  
  
    private fun poidsEnMoins(d : Int) // possible = utilitaire  
        = (d / 1000.0) / 1000  
}
```

- L'attribut `nom` est `public` (par défaut) : accessible en lecture/écriture
- L'attribut `age` est `private` : aucun accès possible
- L'attribut `race` est `public`, mais immuable : accessible en lecture
- L'attribut `poids` est restreint en écriture : accessible en lecture
- La fonction `courir()` est `public` (par défaut) : accessible
- La fonction `poidsEnMoins()` est `private` : aucun accès possible

d'UML à Kotlin : processus systématique



- Simple **ré-écriture** pour les **classes**, les **attributs** (type, visibilité), les **méthodes** (signature, visibilité)

- **Quid des associations ?**

- ▶ Une association **unidirectionnelle** devient un **attribut** dans la classe "source"
- ▶ Une association **bidirectionnelle** devient **deux attributs**, un de chaque côté de l'association
- ▶ Les **rôles** deviennent les **noms** des attributs
- ▶ (ajouter des méthodes pour **mettre à jour** les nouveaux attributs)
- ▶ Les **cardinalités** **0..1** donnent des variables **nullable ?**
- ▶ Les **cardinalités** **0..***, **1..*** ou ***** donnent des **Array?<X>** (ou d'autres collections)

```

class Citoyen (nom : String, prenom : String) {
    private var nom : String
    private var prenom : String
    private var numElecteur : Int
    private var bureau : BureauDeVote?

    init {
        this.nom = nom
        this.prenom = prenom
        this.numElecteur = 0
        this.bureau = null
    }

```

```

fun modifierNumElecteur(nouveauNum : Int) {
    // TODO
}

```

```

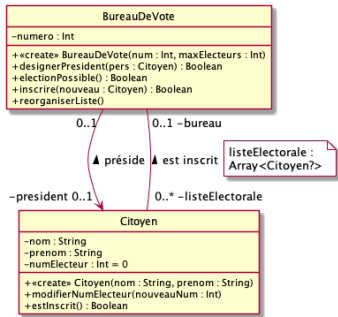
fun modifierBureauDeVote(nouveauBureau : BureauDeVote) {
    // TODO
}

```

```

fun estInscrit() = {
    // TODO
}

```



```

class BureauDeVote (num : Int, maxElecteurs : Int) {
    private val numero : Int
    private var president : Citoyen?
    private val listeElectorale : Array<Citoyen?>

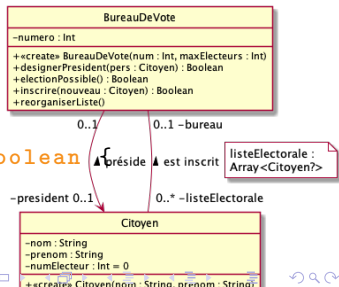
    init {
        numero = num
        president = null
        listeElectorale = arrayOfNulls<Citoyen>(maxElecteurs)
    }

    fun designerPresident(pers : Citoyen) {
        // TODO
    }

    fun electionPossible() {
        // TODO
    }

    fun inscrire(nouveau : Citoyen) : Boolean {
        // TODO
    }
}

```



- 1 Les bases du langage
- 2 Classes et objets
- 3 Variables == références**
- 4 L'héritage
- 5 Les collections
- 6 Les exceptions

Les variables sont des références (nullable)

- En Kotlin, les "variables" sont des **références** (dans la pile mémoire) qui "pointent" vers leur valeur (dans le tas mémoire)
 - ▶ On ne s'occupe pas de réserver de l'espace mémoire
 - ▶ On ne gère pas non plus la libération de cet espace ==> le *Garbage Collector* (=ramasse-miette)
- Si variable = **référence** alors elle peut "pointer" vers rien ?
En Kotlin, **NON** sauf si on a précisé **explicitement** qu'elle pouvait.
- ▶ Ajouter `?` après le type indique que la variable est **possiblement** `null`
- ▶ idem pour les paramètres et/ou le résultat d'une fonction

```
var w : Int
val x : Int?
var y : Double? = 10.0
var z : String? = "totoro"
// w = null
// erreur de compilation
y = null
z = null
```


Utiliser des variables *nullable*

Kotlin **verrouille** l'accès aux variables *nullable*.

- 1 Réaliser des appels "sûrs" via `?` :
`z?.length` retourne `z.length` si `z` \neq `null` sinon retourne `null`
- 2 Utiliser l'opérateur Elvis `?:`
`z?.length ?: 0` : si la partie gauche, ici `z?.length`, $=$ `null` alors on retourne la partie droite, ici `0`
- 3 Forcer l'évaluation via `!!` :
`z!!` retourne une version non-nulle de `z` si `z` \neq `null` mais si `z` $=$ `null`
NullPointerException

```
var z : String? = "totoro"
...
//val l = z.length
// erreur de compilation

var l = z?.length
println(l)

l = z!!.length
println(l)

l = z?.length ?: 0
println(l)
```

- 1 Les bases du langage
- 2 Classes et objets
- 3 Variables == références
- 4 L'héritage**
- 5 Les collections
- 6 Les exceptions

Héritage en programmation objet

La notion d'héritage est centrale en conception et programmation objet. Elle permet de

- mieux **appréhender** le domaine métier modélisé
 - ▶ *qu'est-ce qui est commun ? qu'est-ce qui est spécifique ?*
 - ▶ **généralisation vs. spécialisation**
- **mutualiser** des parties du code pour éviter la duplication
 - ▶ **covariance**
- mieux **architecturer** le code
- faciliter l'**évolution** du code, la maintenance
- faciliter la **réutilisation** et l'adaptation du code
 - ▶ **polymorphisme**

Déclarer un héritage en Kotlin

```
open class Animal(nom:String, age:Int) {  
    protected var nom :String  
    protected var age : Int  
    private var maitre : Personne?  
  
    init {  
        this.nom = nom  
        this.age = age  
        this.maitre = null  
    }  
  
    fun repondre(unNom : String) =  
        (nom == unNom)  
}
```

```
class Chien(nom:String, age:Int, race:String)  
    : Animal(nom, age) {  
    private val race : String  
  
    init {  
        this.race = race  
    }  
  
    fun aboyer() {  
        println("$nom dit : ouaf ouaf !!!")  
    }  
}
```

- La super-classe **autorise** l'héritage : `open`
 - ▶ Attributs `private` ou `protected` ou `public`
- La sous-classe **déclare** l'héritage via `:` suivi d'un appel au **constructeur** de la super-classe
 - ▶ Les attributs de la super-classe ne sont **pas redéclarés**
 - ▶ La sous-classe **accède uniquement** aux attributs `protected` de la super-classe

Polymorphisme en Kotlin

Polymorphisme

Le **polymorphisme** consiste à **redéfinir** dans une sous-classe l'implémentation d'une méthode définie dans la **super-classe**.

En cas de **covariance**, c'est bien la méthode **redéfinie** de la sous-classe qui est appelée.

- La super-classe déclare les méthodes **autorisées** à être redéfinies : `open`
- La sous-classe déclare les méthodes qu'elle **redéfinie** : `override`
- Dans l'implémentation d'une méthode **redéfinie**, il est possible d'**appeler** la méthode de la super-classe : `super.maMethode()`

Polymorphisme en Kotlin : exemple

```
open class Animal(nom:String,age:Int){
    ...
    open fun ageHumain() : Int {
        return 0
    }

    open fun courir() {
        println("$nom court !!!!")
    }
}
```

```
class Chat(..., pedigree:String)
: Animal(nom, age) {
    ...
    override fun ageHumain():Int{
        return age * 6
    }
}
```

```
class Chien(..., race:String)
: Animal(nom, age) {
    ...
    override fun ageHumain():Int{
        return age * 7
    }

    override fun courir(){
        aboyer()
        super.courir()
        aboyer()
        aboyer()
    }
}
```

- **Animal** autorise la redéfinition de **ageHumain()** et de **courir()**
- **Chien** redéfinit **ageHumain()** et **courir()**
- **Chat** ne redéfinit que **ageHumain()**

Classes abstraites en Kotlin

```
abstract class Animal(nom:String,age:Int){
protected var nom :String
protected var age : Int
private var maitre : Personne?

init {
    this.nom = nom
    this.age = age
    maitre = null
}

fun repondre(unNom : String) =
    (nom == unNom)

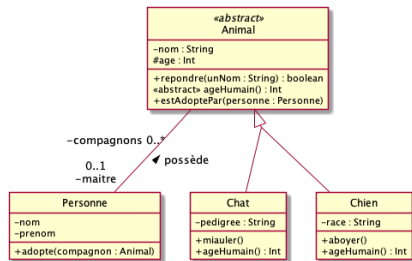
fun estAdoptePar(p : Personne) {
    maitre = p
}

abstract fun ageHumain() : Int

open fun courir() {
    println("$nom court !!!!")
}
```

- La Classe est déclarée **abstraite** par **abstract**
- La classe **déclare** des attributs
- La classe a un **constructeur**
- La classe **déclare** des méthodes (sans proposer d'implémentation) : **abstract**
- La classe **implémente** certaines méthodes
- La classe **autorise** la redéfinition de méthodes : **open**

Héritage : d'UML à Kotlin



```
class Chien(...)
    : Animal(...) {
private val race : String
...
fun aboyer() {
    println("ouaf ouaf !!!")
}
override fun ageHumain() : Int {
    return age * 7
}
```

```
abstract class Animal(...) {
private var nom : String
protected var age : Int
private var maitre : Personne?
...
fun repondre(unNom : String) =
    (nom == unNom)

abstract fun ageHumain() : Int

fun estAdoptePar(p : Personne) {
    maitre = p
}
```

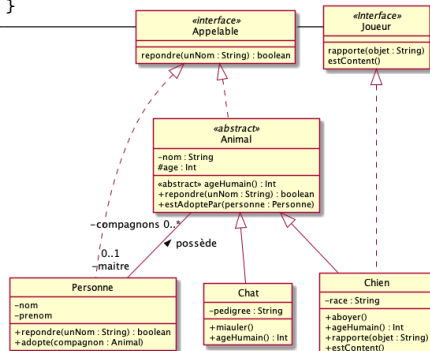
```
class Personne (...) {
private val nom : String
private val prenom : String
private val compagnons : Array<Animal?>
private var nbCompagnons : Int
...
fun adopte(compagnon : Animal) {
    if (nbCompagnons < compagnons.size) {
        compagnons[nbCompagnons]
            = compagnon
        nbCompagnons++
    }
}
```


Interfaces en Kotlin

- Une interface se déclare via **interface**
- Elle déclare des méthodes
 - ▶ Elle peut proposer une implémentation par défaut
- La classe réalisant l'interface l'indique via **:**

```
interface Joueur {  
    fun rapporte(objet : String)  
  
    fun estContent() {  
        println(" :-) ")  
    }  
}
```

```
class Chien(nom:String,age:Int,race:String)  
    : Animal(nom,age), Joueur {  
    ...  
    override fun rapporte(objet : String) {  
        courir()  
        print("$nom rapporte $objet")  
        if (maitre != null)  
            print(" a ${maitre!!.donneNom()}")  
        println("")  
    }  
}
```



- 1 Les bases du langage
- 2 Classes et objets
- 3 Variables == références
- 4 L'héritage
- 5 Les collections
 - Les tableaux de taille fixe
 - Les collections dynamiques
- 6 Les exceptions

Tableaux de taille fixe

1 déclarer un tableau prérempli

```
val notes = arrayOf(12.0, 7.0, 10.5, 8.2, 17.8)
val matieres = arrayOf("Info", "Math", "Anglais", "Eco", "Comm")
```

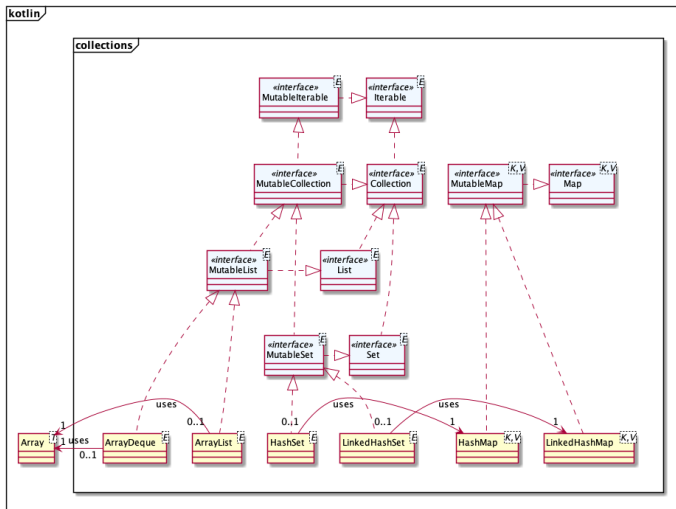
2 déclarer un tableau vide

```
val notes0 = arrayOfNulls<Double>(4)
val matieres0 = arrayOfNulls<String>(10)
```

- dans le cas 2. il faut déclarer le **type** des éléments contenus `<...>` et la **taille** du tableau
- dans le cas 2. toutes les cases du tableau contiennent la valeur `null`²
- le type des tableaux est `Array<Double?>` et `Array<String?>`
- la **taille** du tableau est **définitivement fixée**

2. On y reviendra

Package `kotlin.collections`



<https://kotlinlang.org/docs/collections-overview.html#collection-types>

List<E> et MutableList<E>

collections **ordonnées**, **doublons possible**

- `ArrayList` = implémentation de l'interface `MutableList<E>`
- utilise un `Array<E?>` sous-jacent
 - ▶ procède à des **redimensionnements** automatiques
- Les fonctions
 - ▶ `listOf(..) : List<E>` et
 - ▶ `mutableListOf(..) : MutableList<E>`

instancient de manière **sous-jacente** un objet de type `ArrayList<E>`

- Constructeurs possible :
 - ▶ `ArrayList<E>()`,
 - ▶ `ArrayList<E>(initialCapacity : Int)` ou
 - ▶ `ArrayList<E>(elements : Collection<E>)`

Set<E> et MutableSet<E>

collections **sans doublon**, ordre non garanti

HashSet et LinkedHashSet = **implémentations** possible de l'interface
MutableSet<E>

- HashSet<E> ne préserve pas l'ordre d'insertion
- LinkedHashSet<E> préserve l'ordre d'insertion, mais c'est plus coûteux

Les deux implémentations sont basées sur des **tables de hachage** pour détecter efficacement les **doublons**

- utilisent la fonction hashCode() de K

Les fonctions

- setOf(..) : Set<E> et
- mutableSetOf(..) : MutableSet<E>

instancient de manière **sous-jacente** un objet LinkedHashSet<E>

- 1 Les bases du langage
- 2 Classes et objets
- 3 Variables == références
- 4 L'héritage
- 5 Les collections
- 6 Les exceptions**

Capturer des exceptions

```
val prenom = arrayOf<String>(  
    "Jean-Francois", "Ali",  
    "Christine", "Jean-Francois", "Arnaud")  
  
fun acces(pos : Int, div : Int)  
= prenom[pos/div]  
}  
  
fun main() {  
    val indice = 8  
    val facteur = 0  
    var prenom = ""  
    try {  
        prenom += acces(indice, facteur)  
        println("### ok")  
    }  
    catch (e: ArithmeticException) {  
        println("*** Erreur : $e ***")  
        prenom += acces(indice, 2)  
    }  
    catch (e: ArrayIndexOutOfBoundsException) {  
        println("*** Erreur : $e ***")  
        prenom += acces(0, 1)  
    }  
    finally {  
        println("Prenom : $prenom")  
    }  
}
```

- le bloc `try` englobe le code à risque ; son exécution est **interrompue** dès la survenue d'une exception
- le bloc `catch` est exécuté s'il correspond à l'exception levée
- si aucun bloc `catch` ne correspond à l'exception alors elle est **remontée**
- le bloc optionnel `finally` est exécuté à la suite dans tous les cas

Lever une exception

Pour lever une exception il suffit d'utiliser l'instruction `throw` suivie d'une exception.

Exécuter une instruction `throw` provoque l'interruption instantanée du code, et

- remonte jusqu'à un bloc `try... catch` correspondant à l'exception
- ou provoque la terminaison du programme

Exemple

```
fun acces(indice : Int) : String {  
    if (indice < 0 || indice >= prenom.size)  
        throw IllegalArgumentException("$indice")  
    return prenom[indice]  
}
```

Lever une exception

Pour lever une exception il suffit d'utiliser l'instruction `throw` suivie d'une exception.

Exécuter une instruction `throw` provoque l'interruption instantanée du code, et

- remonte jusqu'à un bloc `try... catch` correspondant à l'exception
- ou provoque la terminaison du programme

Exemple

```
fun acces(indice : Int) : String {  
    if (indice < 0 || indice >= prenom.size)  
        throw IllegalArgumentException("$indice")  
    return prenom[indice]  
}
```