

R3.04 : Qualité de développement

Nouveautés Kotlin : classes anonymes et λ -expressions

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



IUT Nantes
Pôle Sciences et technologie

Nantes Université

Département informatique

- 1 Les classes anonymes en Kotlin
- 2 λ -expressions et fonctions d'ordre supérieur
- 3 Documenter du code Kotlin avec KDoc

Les classes anonymes : implémentation à la volée

Il est possible d'implémenter "à la volée" une classe abstraite ou une interface : il est bien entendu nécessaire d'implémenter les méthodes abstraites nécessaires

```
abstract class Animal(  
    var nom : String) {  
  
    fun appeler(unNom : String)  
        = (nom == unNom)  
  
    abstract fun deplacer()  
}
```

```
class Chat(nom : String)  
    : Animal(nom) {  
    override fun deplacer() {  
        println("grimpe, griffe")  
    }  
}
```

```
val unchat = Chat("Gaga")  
chat.appeler("Gaga")  
chat.deplacer()  
  
val animalInconnu = object : Animal("xxx") {  
  
    override fun deplacer() {  
        println("cours, vole, saute !!!!")  
    }  
}  
  
animalInconnu.appeler("Gaga")  
animalInconnu.deplacer()
```

Classes anonymes : exemple en JavaFX

Ajout d'un "écouteur" à un composant graphique (ici, en JavaFX) :

```
class MonEcouteur(val txt : TextField)
    : EventHandler<ActionEvent> {
    override
    fun handle(event: ActionEvent?) {
        txt.text = "ok"
    }
}
```

```
val unBouton = Button("Go")
val unTexte = TextField()
unBouton.addEventHandler(
    ActionEvent.ACTION,
    MonEcouteur(unTexte)
)
```

peut se remplacer par

```
val unBouton = Button("Go")
val unTexte = TextField()
unBouton.addEventHandler(ActionEvent.ACTION,
    object : EventHandler<ActionEvent> {
        override fun handle(event: ActionEvent?) {
            unTexte.text = "ok"
        }
    }
)
```

Classes anonymes : exemple de `Comparator<X>`

Implémentation "à la volée" d'un critère de tri "spécifique" :

```
val chiens = listOf(Chien("Potter", age = 4),
    Chien("Rogue", Race.BergerAustralien, 2),
    Chien("Tequila", Race.BergerAustralien, 1),
    Chien("Janus", Race.BouvierBernois, 8))

val chiensTries = chiens.sortedWith(
    object : Comparator<Chien> {
        override fun compare(o1: Chien?, o2: Chien?): Int {
            return (o1?.age ?: 0) - (o2?.age ?: 0)
        }
    }
)
```

Classes anonymes : encore un exemple

```
interface Operation {  
    fun calcule(x : Int) : Int  
}
```

```
fun Array<Int>.applique(  
    ope : Operation) {  
    for (i in 0..this.size-1)  
        this[i] = ope.calcule(this[i])  
}  
  
fun Array<Int>.affiche() {  
    for (valeur in this)  
        print("$valeur ")  
    println()  
}
```

- méthodes d'extension
- + classes anonymes

```
val valeurs = arrayOf(3, 10, 5, -7,  
                      9, 12, -1)  
  
valeurs.affiche()  
  
valeurs.applique(object : Operation {  
    override fun calcule(x: Int): Int {  
        return x + 1  
    }  
})  
valeurs.affiche()  
  
valeurs.applique(object : Operation {  
    override fun calcule(x: Int): Int {  
        return ((2 * x) - 10) / 2  
    }  
})  
valeurs.affiche()
```

```
3 10 5 -7 9 12 -1  
4 11 6 -6 10 13 0  
-1 6 1 -11 5 8 -5
```

Pour aller plus loin

- classes internes `inner`,
- classes imbriquées,
- classes scellées `sealed`,
- La généricité
- ...

- 1 Les classes anonymes en Kotlin
- 2 λ -expressions et fonctions d'ordre supérieur
- 3 Documenter du code Kotlin avec KDoc

λ-expression

Une λ-expression (lire "lambda-expression") est une **fonction** (au sens **mathématique**) , càd une "expression Kotlin" définissant

- des paramètres éventuels (et leurs types)
- un traitement
- un résultat éventuellement renvoyé (et son type)

```
val f : (Int, Int) -> Int = { x, y -> x*x + 3 * y + 2 }
```

correspond à

$$f : x, y \mapsto x^2 + 3y + 2$$

- Une λ-expression peut-être stockée dans une variable `val` ou `var`
- Le type d'λ-expression peut-être inféré
- `Unit` est utilisé comme type si aucun résultat renvoyé

Exemples de λ -expressions

```
val f2 = { x : Int, y : Int -> x*x + 3 * y + 2 }

val afficheNom : (String) -> Unit =
  {nom -> println("Hello ${nom.toUpperCase()}!!!") }

val affichage = { println("Hello students !!!") }

val addition = { x: Int, y: Int ->
  val res = x + y
  println("$x + $y = $res")
  res }

val utiliser = { chien: Chien, nom: String ->
  chien.nommer(nom)
  chien.appeler("totoro")
  println(chien.race) }
```

Utilisation :

```
f(3, 4)
val r = f2(10, 0)
affichage()
var y = addition(r, 10)
utiliser(Chien(Race.Beauceron,10), "Potter")
```

Fonctions d'ordre supérieur

Fonction d'ordre supérieur

Une fonction (ou une méthode) d'ordre supérieur prend **en paramètre** une (ou des) **fonction(s)** :

- la fonction sera notée `inline`
- les paramètres fonctionnels seront des **λ -expressions**

```
inline fun app(x : Int, y : Int, comp : (Int, Int) -> Boolean) : Boolean {  
    return comp(x,y)  
}
```

```
app(3, 4, {a, b -> a == b})  
app(3, 4, {x, y -> x > y})  
app(6, -3, {x, y -> x > y})  
app(6, 10, { z1, z2 ->  
    val r = z1*z1 + 5 * z2 +10  
    r == 0  
})
```

Fonctions d'ordre supérieur (2)

```
inline fun app(x : Int, y : Int, comp : (Int, Int) -> Boolean) : Boolean {  
    return comp(x,y)  
}
```

Lorsque le **dernier paramètre** d'une fonction est une **λ -expression**, il est possible de l'écrire en dehors des parenthèses

```
app(3, 4) { a, b -> a == b }  
app(3, 4) { x, y -> x > y }  
app(6, -3) { x, y -> x > y }  
app(6, 10) {  
    z1, z2 ->  
    val r = z1*z1 + 5 * z2 +10  
    r == 0  
}
```

Ok...mais à quoi ça sert???

Fonctions d'ordre supérieur (2)

```
inline fun app(x : Int, y : Int, comp : (Int, Int) -> Boolean) : Boolean {  
    return comp(x,y)  
}
```

Lorsque le **dernier paramètre** d'une fonction est une **λ -expression**, il est possible de l'écrire en dehors des parenthèses

```
app(3, 4) { a, b -> a == b }  
app(3, 4) { x, y -> x > y }  
app(6, -3) { x, y -> x > y }  
app(6, 10) {  
    z1, z2 ->  
    val r = z1*z1 + 5 * z2 +10  
    r == 0  
}
```

Ok...mais à quoi ça sert ???

Exemple `List<T>.filter()`

`List<T>.filter(predicate : (T) -> Boolean)` prend en paramètre une λ -expression `predicate` définissant la règle de filtrage

```
val chiens = listOf(Chien("Potter", age = 4),
    Chien("Rogue", Race.BergerAustralien, 2),
    Chien("Tequila", Race.BergerAustralien, 1),
    Chien("Janus", Race.BouvierBernois, 8))

val jeunesChiens = chiens.filter { chien -> chien.age <= 4 }

val bergersAustraliens = chiens.filter { it.race == Race.BergerAustralien }

val autresChiens = chiens.filterNot { it.race == Race.BergerAustralien }

//val rogues = chiens.filter{it.nom == "Rogue"}
```

- Si on ne précise pas le paramètre de la λ -expression, par défaut `it`
- L'encapsulation est **préservée** : on n'accède qu'aux attributs `public`
- <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections.filter.html>

Autres exemples dans `List<T>`

```
// on precise le parametre de la fonction : ici chien
chiens.forEach { chien -> chien.aboyer() }
chiens.all { chien -> chien.age < 20 }

// on ne precise pas le parametre
// par default, parametre = it
chiens.count { it.race == Race.BergerAustralien }
chiens.maxOf { it.age }
```

- `List<T>.forEach(action : (T) -> Unit)` : applique `action` à tous les éléments
- `List<T>.all(predicate : (T) -> Boolean)` : `true` si tous les éléments vérifient `predicate`
- `List<T>.count(predicate : (T) -> Boolean)` : le nombre d'éléments qui respectent le `predicate`
- `List<T>.maxOf(selector : (T) -> Double)` : retourne le plus grand des éléments suivant le critère donné par `selector`
- ...

Interfaces fonctionnelles

Interface fonctionnelle

Une interface **fonctionnelle** est une interface qui peut être **réalisée** par une λ -expression (à la place d'une classe anonyme) :

- = **Interface SAM** (Single Abstract Method)
- Il faut ajouter le mot-clef `fun` dans la déclaration de l'interface
- L'interface ne peut déclarer **qu'une unique méthode**

```
fun interface Appelleable {  
    fun appeler(nom : String)  
        : Boolean  
}
```

```
val animal = object : Appelleable {  
    override fun appeler(nom: String): Boolean  
        return nom == "Totoro"  
    }  
val animal2 = Appelleable {  
    nom -> nom == "Totoro" }  
  
// si un seul parametre, on peut l'omettre  
val animal3 = Appelleable { it == "Totoro" }
```

L'intérêt des interfaces fonctionnelles est de **simplifier grandement** le code dans le cas d'**implémentation à la volée** de ces interfaces

Interfaces fonctionnelles : exemple (1)

```
unBouton.addHandler(ActionEvent.ACTION,
    object : EventHandler<ActionEvent> {
        override fun handle(event: ActionEvent?) {
            texte.text = "ok"
        }
    }
)

// peut s'écrire beaucoup plus simplement ainsi

unBouton.addHandler(ActionEvent.ACTION) { texte.text = "ok" }
```

La capture d'événements en **Kotlin/Android** se fait très souvent via des implémentations d'interfaces fonctionnelles

```
val bouton = findViewById<Button>(R.id.main_bouton)

bouton.setOnClickListener {
    // traitement lors du clic sur un bouton
    // ...
    System.out.println("clic sur le bouton")
}
```

Interfaces fonctionnelles : exemple (1)

```
unBouton.addHandler(ActionEvent.ACTION,
    object : EventHandler<ActionEvent> {
        override fun handle(event: ActionEvent?) {
            texte.text = "ok"
        }
    }
)

// peut s'écrire beaucoup plus simplement ainsi

unBouton.addHandler(ActionEvent.ACTION) { texte.text = "ok" }
```

La capture d'événements en **Kotlin/Android** se fait très souvent via des implémentations d'interfaces fonctionnelles

```
val bouton = findViewById<Button>(R.id.main_bouton)

bouton.setOnClickListener {
    // traitement lors du clic sur un bouton
    // ...
    System.out.println("clic sur le bouton")
}
```

Interfaces fonctionnelles : exemple (2)

```
val chiens = listOf(Chien("Potter", age = 4),
    Chien("Rogue", Race.BergerAustralien, 2),
    Chien("Tequila", Race.BergerAustralien, 1),
    Chien("Janus", Race.BouvierBernois, 8))

val chiensTries = chiens.sortedWith(object : Comparator<Chien> {
    override fun compare(o1: Chien?, o2: Chien?): Int {
        return (o1?.age ?: 0) - (o2?.age ?: 0)
    }
})

// peut s'ecrire beaucoup plus simplement ainsi

val chiensTries2 = chiens.sortedWith { o1, o2 ->
    (o1?.age ?: 0) - (o2?.age ?: 0) }
```

- 1 Les classes anonymes en Kotlin
- 2 λ -expressions et fonctions d'ordre supérieur
- 3 Documenter du code Kotlin avec KDoc

Documentation Kotlin : KDoc

Des commentaires **particuliers** peuvent être ajoutés au code Kotlin afin de générer une documentation sous la forme de fichiers HTML : **le langage KDoc**

```
/**
 * Defini un objet representant un *Chien*.
 * @property race la race du chien.
 * @constructor construit un chien d'une race donnee sans le nommer.
 */
class Chien(private val race : Race) {
    private var nom = ""
    /**
     * Fonction permettant de *nommer* un chien.
     * @property nouveau le nouveau nom du chien.
     */
    fun nommer(nouveau : String) {
        nom = nouveau
    }
}
```

- Les blocs KDoc **commencent** avec un `/**` et **terminent** avec `*/`.
- Toutes les lignes **commencent** avec une `*` et **terminent** avec un `.`

Annotations KDoc utilisables

Des @notations particulières permettent de préciser certains éléments

- `@constructor` – documente le constructeur primaire d'une classe
- `@property name` – documente un attribut **public** de la classe
- `@param param` – documente
 - ▶ un paramètre d'une fonction
 - ▶ un paramètre d'une classe
 - ▶ un paramètre d'un constructeur secondaire
- `@return` – documente la valeur de retour de la fonction
- `@throws class` – documente une exception possiblement levée par la fonction
- `@author` – documente l'auteur du code
- `@since` – documente la version du code
- `@see identifier` – ajoute un lien vers un élément identifié par `identifier` dans une section **See also**

Exemple KDoc complet

```
/**
 * Defini un objet representant un *Chien*.
 *
 * @property race la race du chien.
 * @property nom le nom du chien.
 * @param race la race du chien.
 * @constructor construit un chien d'une race donnee sans le nommer.
 * @see Race les races possibles de chien.
 * @author A. Lanoix.
 * @since 0.42.
 */
class Chien(val race: Race = Race.Beauceron) {
    var nom = ""

    /**
     * Construit un chien d'une race donnee, qui a un nom des sa naissance.
     *
     * @param nom le nom du chien.
     * @param race la race du chien.
     */
    constructor(nom: String, race: Race) : this(race) {
        this.nom
    }
}
```

```

/**
 * Fonction permettant de *nommer* un chien.
 *
 * @param nouveau le nouveau nom du chien.
 */
fun nommer(nouveau: String) {
    nom = nouveau
}

/**
 * Fonction donnant le nom du chien en majuscule.
 *
 * @return le nom du chien en majuscule.
 */
fun donnerLeNom() = nom.uppercase()

/**
 * Fonction essayant d'appeler le chien.
 *
 * @param unNom le nom appele.
 * @return [true] si [unNom] correspond au nom du chien, [false] sinon.
 */
fun appeler(unNom: String): Boolean {
    return (nom == unNom)
}

```


Documentation HTML générée grâce à Dokka

The screenshot shows the Dokka-generated HTML documentation for the `Chien` class. The interface includes a dark header with the project name "CMs 1.0-SNAPSHOT" and a search icon. A left sidebar displays a navigation tree with "CMs" expanded, showing "[root]", "cm2", and "Chien" (which is selected). The main content area shows the class name "Chien" in large font, followed by the class signature `class Chien(race: Race)`. Below this, a description states: "Défini un objet représentant un `cm2.Chien`." The "Author" is listed as "A. Lanoix." and the "Since" version is "0.42". A horizontal menu contains tabs for "Constructors", "Functions", "Properties", "Parameters", and "See also", with "Constructors" being the active tab. Under the "Constructors" tab, two constructors are listed: 1. `fun Chien(nom: String, race: Race)` with the description "Construit un chien d'une race donnée, qui a un nom dès sa naissance." 2. `fun Chien(race: Race = Race.Beauceron)` with the description "construit un chien d'une race donnée sans le nommer."

<https://github.com/Kotlin/dokka>

Utiliser Dokka

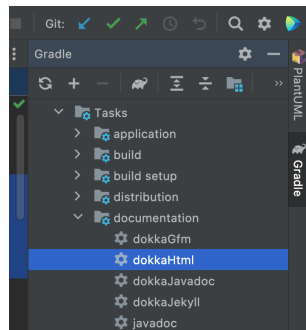
- 1 Ajoutez le plugin gradle **Dokka** au fichier `build.gradle.kts` :

```
plugins {  
    kotlin("jvm") version "1.x.xx"  
    id("org.jetbrains.dokka") version "1.x.xx"  
    application  
}
```

- 2 puis générez la documentation via le terminal :

```
./gradlew dokkaHtml
```

ou via IntelliJ



La documentation se trouve ensuite dans `build/dokka/html/`