

# R3.04 : Design patterns

## Nouveautés Kotlin : à propos des classes

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



**IUT Nantes**  
Pôle Sciences et technologie

**Nantes Université**

Département informatique

- 1 A propos des constructeurs en Kotlin
- 2 Éléments statiques d'une classe Kotlin
- 3 Initialisation retardée des variables `var` et `val`
- 4 classes particulières : enumeration, classes de données et extension de classes

# Rappel : constructeur en Kotlin

```
class Chien (monNom : String, race : String = "inconnue", poids : Double) {  
    var nom :String  
    private var age : Int = 1  
    private val race : String  
    var poids : Double  
        private set  
  
    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
- certains attributs peuvent être **immuables** `val`
- On peut restreindre la visibilité : `private` ou `private set`

# Constructeur primaire "simplifié"

Il est possible de déclarer les attributs de la classe directement dans le constructeur

## Attention

Possible UNIQUEMENT si attribut = paramètre du constructeur

```
class Chien(var nom :String,  
            private var age : Int = 1,  
            private val race : String = "inconnue",  
            poids : Double) {  
  
var poids : Double private set  
  
init {  
    this.poids = if (poids > 0.0) poids else 0.1  
}
```

Toujours possible

- Valeurs par défaut
- Attributs immuables : `val`
- Modification de la visibilité : `private`

# Constructeurs secondaires

Il est possible d'ajouter d'autres constructeurs (quand c'est nécessaire)

```
class Chien private constructor(var nom :String,  
    private var age : Int = 1,  
    private val race : String = "inconnue",  
    poids : Double) {  
    ...  
  
    constructor(poids : Double) :  
        this(nom = "xxx", poids = poids) {  
        // TODO  
    }  
}
```

- Ici le constructeur "primaire" est **privé** (pas obligatoire)
- Un constructeur doit **toujours** rappeler un autre constructeur via

`this(...)`

- 1 A propos des constructeurs en Kotlin
- 2 Éléments statiques d'une classe Kotlin
- 3 Initialisation retardée des variables `var` et `val`
- 4 classes particulières : enumeration, classes de données et extension de classes

# Éléments statiques d'une classe

## Statique

Un attribut ou une méthode **statique** est un élément **commun** à toutes les instances de la classe considérée

- Une méthode statique appartient à la classe mais peut s'utiliser **sans instancier** d'objet
  - ▶ Les attributs (non statique) de la classe ne sont **pas accessibles** depuis une méthode statique
- Un attribut statique est un attribut **partagé** par toutes les instances

## Diagramme de classes UML

Les attributs et les méthodes statiques apparaissent soulignés

# Éléments statiques d'une classe

## Statique

Un attribut ou une méthode **statique** est un élément **commun** à toutes les instances de la classe considérée

- Une méthode statique appartient à la classe mais peut s'utiliser **sans instancier** d'objet
  - ▶ Les attributs (non statique) de la classe ne sont **pas accessibles** depuis une méthode statique
- Un attribut statique est un attribut **partagé** par toutes les instances

## Diagramme de classes UML

Les attributs et les méthodes statiques apparaissent soulignés



# Éléments statiques en Kotlin : le companion object

Kotlin propose le bloc `companion object` {...} dans lequel on déclare les attributs et les méthodes statiques

```
class Chien(  
    val race: Race = Race.Beauceron,  
    var age : Int = 1) {  
    lateinit var nom : String  
  
    init {  
        compteurChien++  
    }  
  
    fun nommer(nouveau: String) {  
        nom = nouveau  
    }  
  
    fun appeler(unNom: String) = (nom == unNom)  
  
    companion object {  
        var compteurChien = 0  
  
        fun afficheTaille(race : Race) {  
            println(race.tailleCourante())  
        }  
    }  
}
```

- Un attribut statique pourrait être `private`
- Observez le bloc `init {...}`
- Usage des éléments statiques :

```
...  
val rogue = Chien(Race.BergerAust  
val potter = Chien(age = 3)  
  
println(Chien.compteurChien)  
  
Chien.afficheTaille(  
    Race.Malinois)
```

# Éléments statiques en Kotlin : le companion object (2)

```
class Chien(...) {  
    companion object {  
        fun lePlusVieux(ch1 : Chien, ch2 : Chien) : Chien? {  
            return if (ch1.age > ch2.age) ch1  
                else if (ch2.age > ch1.age) ch2  
                else null  
        }  
    }  
}
```

Souvent, une méthode statique a en paramètre et/ou résultat des objets du type de la classe

```
val rogue = Chien(Race.BergerAustralien)  
val potter = Chien(age = 3)  
...  
val vieuxChien = Chien.lePlusVieux(rogue, potter)
```

# Sommaire

- 1 A propos des constructeurs en Kotlin
- 2 Éléments statiques d'une classe Kotlin
- 3 Initialisation retardée des variables `var` et `val`
- 4 classes particulières : enumeration, classes de données et extension de classes

# Initialisation "retardée"

Normalement un attribut **doit obligatoirement** prendre une valeur dès l'initialisation. On peut **retarder** l'initialisation d'un attribut `var`, sans pour autant le déclarer nullable : `lateinit`

```
class Chien(val race: Race = Beauceron) {  
    lateinit var nom : String  
  
    fun nommer(nouveau: String) {  
        nom = nouveau  
    }  
    fun appeler(unNom: String) = (nom == unNom)  
}
```

Un attribut `lateinit` doit être initialisé avant d'être utilisé :

```
kotlin.UninitializedPropertyAccessException:  
    lateinit property nom has not been initialized
```

# Initialisation "retardée"

Normalement un attribut **doit obligatoirement** prendre une valeur dès l'initialisation. On peut **retarder** l'initialisation d'un attribut `var`, sans pour autant le déclarer nullable : `lateinit`

```
class Chien(val race: Race = Beauceron) {  
    lateinit var nom : String  
  
    fun nommer(nouveau: String) {  
        nom = nouveau  
    }  
    fun appeler(unNom: String) = (nom == unNom)  
}
```

Un attribut `lateinit` doit être initialisé avant d'être utilisé :

```
kotlin.UninitializedPropertyAccessException:  
    lateinit property nom has not been initialized
```

# Initialisation "paresseuse"

On peut attendre la première utilisation d'un attribut `val` pour l'initialiser. L'initialisation est déclarée dans un bloc `by lazy {...}`.

```
class Chien(val nom : String, var age: Int = 1) {  
  
    val tatouage : String by lazy {  
        println("le chien $nom est enfin tatoue")  
        val aujourd'hui = LocalDateTime.now()  
        "**$nom*$age*$aujourd'hui**"  
    }  
  
    fun uneAnneDePlus() = age++  
}
```

Le bloc `by lazy {...}` peut contenir plusieurs instructions

## lateinit vs. by lazy

- Les attributs déclarés `lateinit` sont toujours des attributs `var`
- Les attributs initialisés `by lazy` sont toujours des attributs immuables

`val`

# Initialisation "paresseuse"

On peut attendre la première utilisation d'un attribut `val` pour l'initialiser. L'initialisation est déclarée dans un bloc `by lazy {...}`.

```
class Chien(val nom : String, var age: Int = 1) {  
  
    val tatouage : String by lazy {  
        println("le chien $nom est enfin tatoue")  
        val aujourd'hui = LocalDateTime.now()  
        "**$nom*$age*$aujourd'hui**"  
    }  
  
    fun uneAnneDePlus() = age++  
}
```

Le bloc `by lazy {...}` peut contenir plusieurs instructions

## lateinit vs. by lazy

- Les attributs déclarés `lateinit` sont **toujours** des attributs `var`
- Les attributs initialisés `by lazy` sont **toujours** des attributs immuables `val`

# Initialisation "paresseuse"

On peut attendre la première utilisation d'un attribut `val` pour l'initialiser. L'initialisation est déclarée dans un bloc `by lazy { ... }`.

```
class Chien(val nom : String, var age: Int = 1) {  
  
    val tatouage : String by lazy {  
        println("le chien $nom est enfin tatoue")  
        val aujourd'hui = LocalDateTime.now()  
        "**$nom*$age*$aujourd'hui**"  
    }  
  
    fun uneAnneDePlus() = age++  
}
```

Le bloc `by lazy { ... }` peut contenir plusieurs instructions

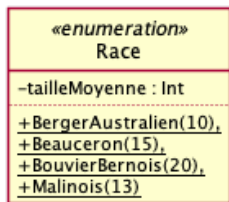
## lateinit vs. by lazy

- Les attributs déclarés `lateinit` sont **toujours** des attributs `var`
- Les attributs initialisés `by lazy` sont **toujours** des attributs immuables `val`



- 1 A propos des constructeurs en Kotlin
- 2 Éléments statiques d'une classe Kotlin
- 3 Initialisation retardée des variables `var` et `val`
- 4 classes particulières : enumeration, classes de données et extension de classes

# Des classes particulières : les énumérations



```
enum class Race(private val tailleMoyenne : Int) {  
    BergerAustralien(10),  
    Beauceron(15),  
    BouvierBernois(20),  
    Malinois(13);  
  
    fun tailleCourante() : Int {  
        return tailleMoyenne  
    }  
}
```

- Une énumération est une **classe** : les valeurs sont des instances de la classe
- Une énumération peut avoir des **attributs** et des **méthodes**
- La méthode (prédéfinie) `Race.values()` renvoie un tableau contenant toute les valeurs de l'énumération

## IntelliJ IDEA

File > New > Kotlin **class**/File, et choisir Enum **class** dans le menu.

# Des classes particulières : les classes de données

Les classes de données **simplifient** l'écriture d'objets métier :  
les méthodes `toString()`, `equals(...)` et `hashCode()` sont  
**automatiquement** présentes

```
data class Chien(  
    val nom : String,  
    private var age : Int = 1) {  
  
    fun uneMethode() {  
        // TODO  
    }  
    ...  
}
```

```
val potter = Chien("Potter", age = 4)  
val rogue = Chien("Rogue")  
println(potter)  
println(rogue)  
println(potter != rogue)
```

```
Chien(nom=Potter, age=4)  
Chien(nom=Rogue, age=1)  
true
```

- Une classe de données doivent **obligatoirement** déclarer tous ses attributs dans son **constructeur principal**.
- Les classe de données sont **compatibles** avec les attributs `val` ou `var`, les valeurs par défaut, la visibilité `private`, ...

# Classes de données vs. héritage

- Impossible d'hériter d'une classe de données
- Une classe de données peut hériter d'une classe (abstraite) sous conditions
  - 1 La classe mère doit déclarer tous ses attributs comme `open` dans un constructeur primaire
  - 2 La classe de données doit redéclarer tous les attributs de la classe mère comme `override`

```
abstract class Animal(  
    open var age : Int) {  
    ...  
}
```

```
data class Chat(  
    val nom : String,  
    override var age : Int)  
        : Animal(age) {  
    ...  
}
```

## IntelliJ IDEA

File > New > Kotlin `class`/File, et choisir Data `class` dans le menu.

# Classes de données vs. héritage

- Impossible d'hériter d'une classe de données
- Une classe de données peut hériter d'une classe (abstraite) sous conditions
  - 1 La classe mère doit déclarer tous ses attributs comme `open` dans un constructeur primaire
  - 2 La classe de données doit redéclarer tous les attributs de la classe mère comme `override`

```
abstract class Animal(  
    open var age : Int) {  
    ...  
}
```

```
data class Chat(  
    val nom : String,  
    override var age : Int)  
        : Animal(age) {  
    ...  
}
```

## IntelliJ IDEA

File > New > Kotlin `class`/File, et choisir Data `class` dans le menu.

# Les méthodes d'extension

On peut souhaiter **ajouter** une méthode à une classe pré-existante, **sans modifier** le source de classe concernée

- On **ne souhaite pas** faire un héritage
- Le code source de la classe à étendre n'est généralement **pas accessible**
- La méthode d'extension est déclarée comme une fonction **précédée** du nom de la classe à étendre :

Dans `Chien.kt`

```
class Chien(  
    val race: Race = Beauceron) {  
    lateinit var nom : String  
  
    fun nommer(nouveau: String) {  
        nom = nouveau  
    }  
    fun appeler(unNom: String) =  
        (nom == unNom)  
}
```

Dans `ChienExt.kt` par exemple

```
fun Chien.designer() {  
    nom.uppercase()  
}
```

```
val rogue = Chien(race)  
rogue.nommer("Rogue")  
println(rogue.designer())
```

- **Attention :** Une méthode d'extension n'a accès qu'aux attributs de la classe