



UNIVERSITÉ
TOULOUSE III
PAUL SABATIER

**KEEP
CALM
AND
<CODE/>**

Programmation orientée objet – Application en Kotlin

Attributs et méthodes de classe

Héritage

Sylvie TROUILHET

Attributs et méthodes de classe

Attribut et méthode d'instance : accessible qu'à travers l'objet

Attribut de classe : attribut dont la valeur est **partagée** par toutes les instances de la classe

Si la valeur est modifiée dans une instance, elle est aussi modifiée pour toutes les autres instances

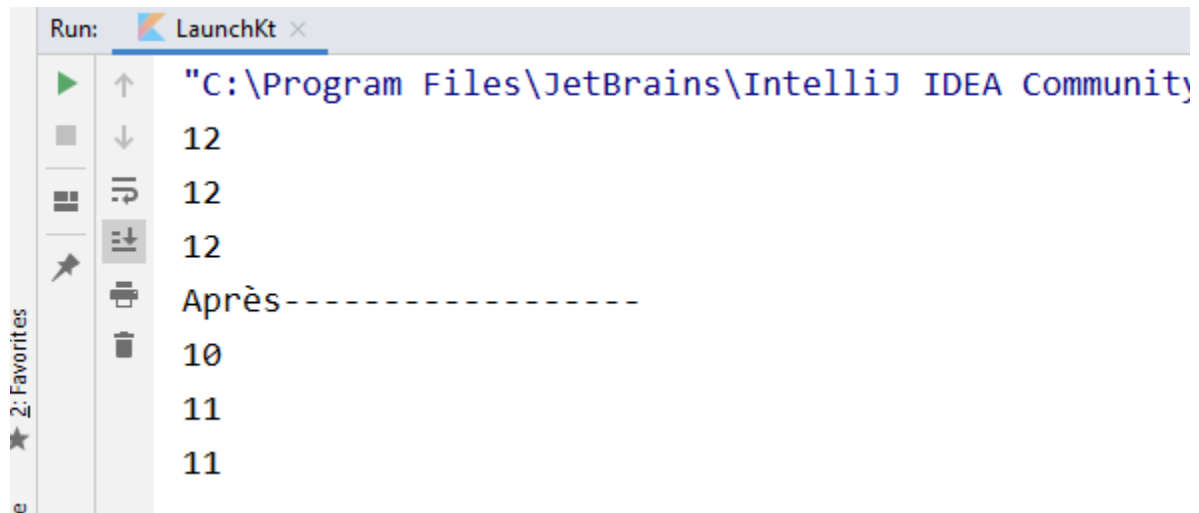
Méthode de classe : méthode s'appliquant à la classe

Exemple

```
fun main() {  
    val p1=Personne();  
    val p2=Personne();  
    println(Personne.nbBonbon)  
    println(p1.nbBonbon)  
    println(p2.nbBonbon)  
    p1.manger()  
    p2.manger()  
    println("Après-----")  
    println(Personne.nbBonbon)  
    println(p1.nbBonbon)  
    println(p2.nbBonbon)  
}
```

```
class Personne {  
    var nbBonbon=12  
    fun manger() {  
        nbBonbon--;  
        Personne.nbBonbon--  
    }  
    companion object {  
        var nbBonbon = 12  
    }  
}
```

Indiquer ce qui est affiché dans le main.



**KEEP
CALM
AND
<CODE/>**

Programmation orientée objet – Application en Kotlin

PLAN Partie 3 : héritage

[Objectifs](#)

[Mise en œuvre en Kotlin](#)

[Héritage versus composition](#)

[Classe de base et dérivées](#)

[Redéfinition de méthodes](#)

[Exercice](#)

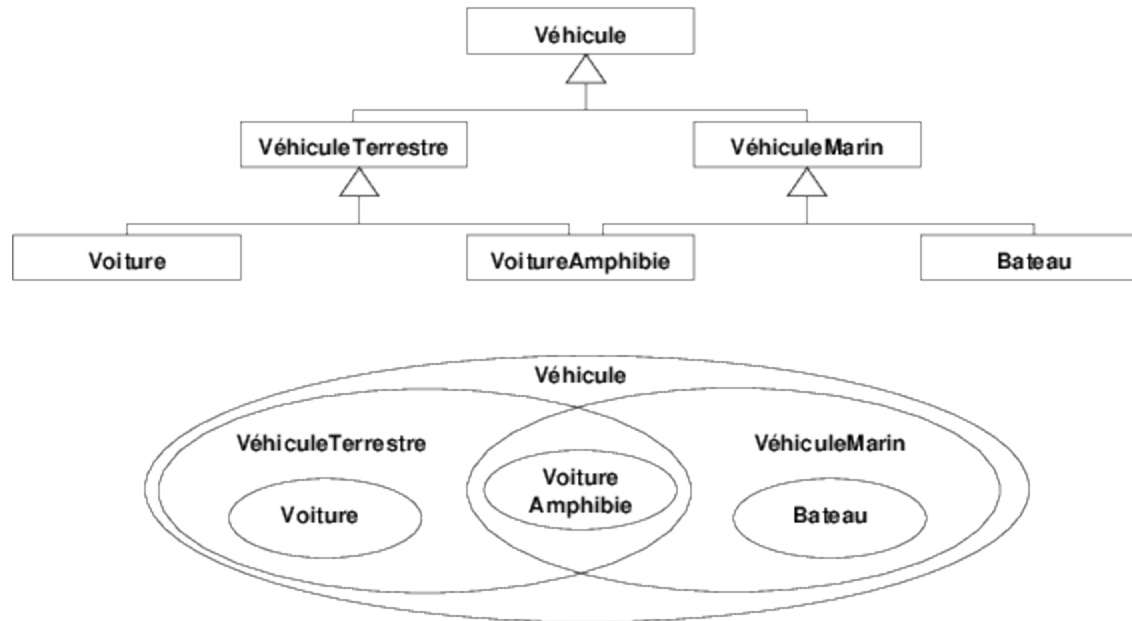
[Compléments](#)

Objectifs de l'héritage

L'héritage permet :

- La **factorisation** d'attributs et de méthodes
- La **réutilisabilité** des composants logiciels (les classes) avec des variantes dans les attributs ou les méthodes
- La **classe fille hérite des attributs et méthodes non privées de sa classe mère** et peut posséder des attributs et/ou des méthodes supplémentaires (qui la spécialisent).

Vision ensembliste



spécialisation / généralisation

Mise en œuvre en Kotlin

Considérons une classe Ville:

- **Attributs** : nom (chaîne - non mutable),
nbHabitants (entier – mutable)
- **Méthodes**: description(): String

```
class Ville(_nom: String, _nbHab: Int) {  
    val nom: String = _nom  
    var nbHab: Int = _nbHab  
  
    fun description(): String =  
        "Ville($nom, $nbHab) "  
}
```


Mise en œuvre en Kotlin

Définissons une classe Capitale qui est une ville particulière :

- Attribut supplémentaire : pays (String – non mutable)

→ La classe Capitale hérite de la classe Ville avec des attributs et méthodes supplémentaires

c'est une **spécialisation** de Ville

Mise en œuvre en Kotlin

```
class Capitale constructor(_nom: String,  
_nbHab: Int, _pays:String)  
    : Ville(_nom, _nbHab) {  
    val pays: String = _pays  
}
```

Par défaut une classe est déclarée **final**

Pour être dérivable, elle doit être déclarée **open**

```
open class Ville(_nom: String, _nbHab: Int) {  
... }
```

Mise en œuvre en Kotlin

```
fun main() {  
val v1=Ville("Castres", 3000)  
    println(v1.description())  
val v2=Capitale("Paris", 2000000, "France")  
    println(v2.description())  
}
```

```
"C:\Program Files\JetB  
Ville(Castres,3000)  
Ville(Paris,2000000)
```

Héritage ou composition ?

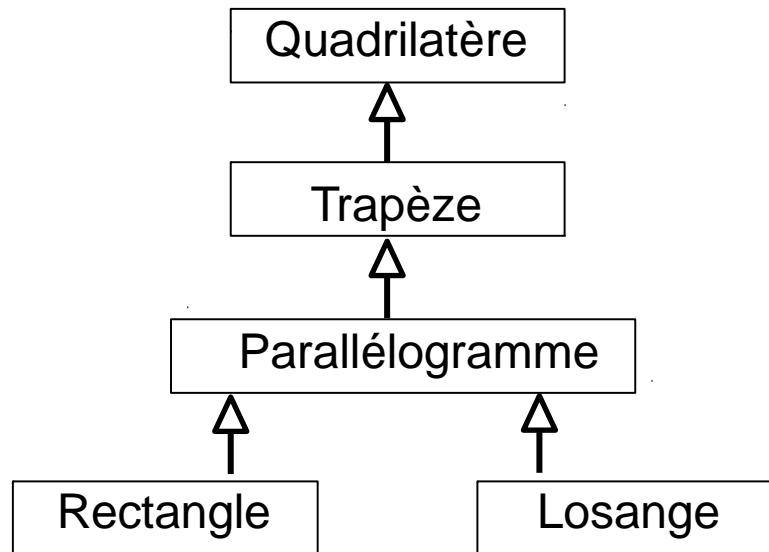
- Une capitale est une Ville particulière donc la classe Capitale hérite de la classe Ville
- Un ordinateur a un écran (MAIS un ordinateur n'est pas un écran particulier) donc la classe Ordinateur n'hérite pas de la classe Ecran
- La classe Ordinateur **est composée** d'un objet de type Ecran et d'autres éléments
- L'écran a une existence indépendante de l'ordinateur

Graphe d'héritage

Kotlin n'autorise que l'**héritage simple** : une classe ne peut hériter directement que d'une autre classe (sinon héritage multiple)

La relation d'héritage simple est transitive et définit un ordre total (les éléments de l'ensemble sont comparables)

Exemple :



Super-type : la classe Any

Toutes les classes héritent d'une classe de base qui contient les opérations applicables sur tous les objets

Kotlin : classe Any

```
public open class Any {  
    public open operator fun equals(other:Any?):Boolean  
    public open fun hashCode() :Int  
    public open fun toString() :String  
}
```

Représente un objet générique que l'on peut comparer et pour lequel on peut donner une description textuelle ou un code (avec une fonction de hachage)

Super-type : la classe Any

Représente un objet générique que l'on peut comparer et pour lequel on peut donner une description textuelle ou un code (avec une fonction de hachage)

Casting intelligent :

```
fun essai(obj:Any) { // obj != null
    if (obj is String) {}
    else {}
}
```

Polymorphisme

Surcharge de méthodes : capacité de définir – dans une même classe – plusieurs méthodes qui possèdent le même nom. Le compilateur choisit la méthode en fonction des paramètres réels.

→ Les méthodes surchargées sont choisies à la **compilation**

Redéfinition de méthodes : une classe fille peut définir une méthode qui possède la même signature (paramètres et type de retour) qu'une méthode de sa classe mère.

→ Les méthodes redéfinies sont choisies à l'exécution – **liaison dynamique**

Redéfinition de méthodes

Reprenons la classe Ville ayant la méthode description

→ comment personnaliser cette méthode pour une capitale pour afficher

`Ville(Paris, 2 000 000) capitale de France`

```
"C:\Program Files\JetB  
Ville(Castres,3000)  
Ville(Paris,2000000)
```

```
fun description(): String =  
    "Ville($nom,$nbHabitants)"
```

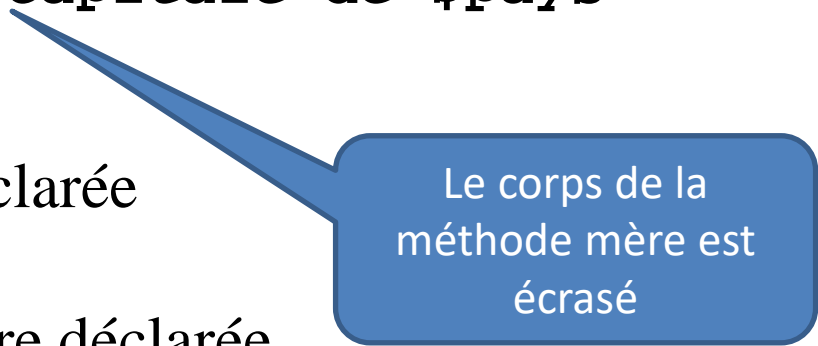
Redéfinition de méthodes

- ☹️ **surtout ne pas modifier la méthode de la classe mère !**
- 😊 Redéfinir la méthode dans la classe fille

```
override fun description(): String =  
    "Ville($nom,$nbHab) capitale de $pays"
```

Par défaut une méthode est déclarée
final

Pour être redéfinie, elle doit être déclarée
open



Le corps de la
méthode mère est
écrasé

Redéfinition de méthode

Pour spécialiser, sans écraser le code de la méthode mère :
appeler la méthode mère avec le mot-clé **super** :

```
override fun description(): String =  
    "${super.description()} capitale de $pays"
```

Exercice

Reprendre la classe `Personne`.

Créer une classe fille `Personnel` ayant un attribut fonction (CDD, CDI, PDG) et redéfinir la méthode `presenteToi()`

```
open class Personne( val nom:String, _age:Int=0) {  
    var age:Int=_age  
    set(_age){if (_age==field+1) field++}  
  
    open fun presenteToi() {  
        println("Je suis $nom et j'ai $age an(s)")  
    }  
}
```

Exercice

Utiliser le main suivant pour tester votre classe. L'affichage doit être le suivant :

```
"C:\Program Files\JetBrains\IntelliJ IDE  
Je suis Toto et j'ai 23 an(s)  
mon statut : CDD
```

```
fun main( ) {  
    val p2=Personnel( "Toto", 23, Statut.CDD)  
    p2.presenteToi( )  
}
```

```
enum class Statut {  
    CDD, CDI, PDG    }
```

Compléments

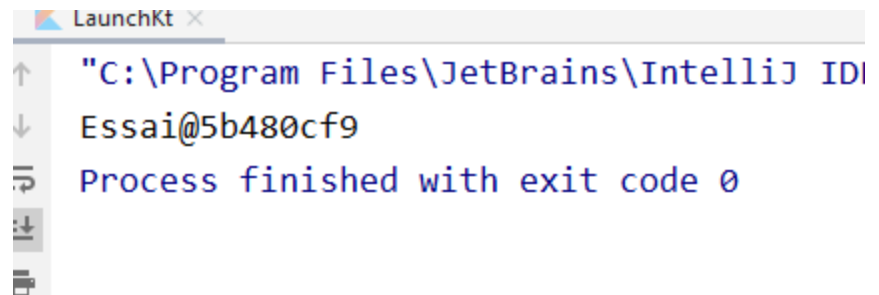
Soit la classe

```
class Essai() {}
```

Peut-on écrire

```
val e=Essai()  
print(e.toString())
```

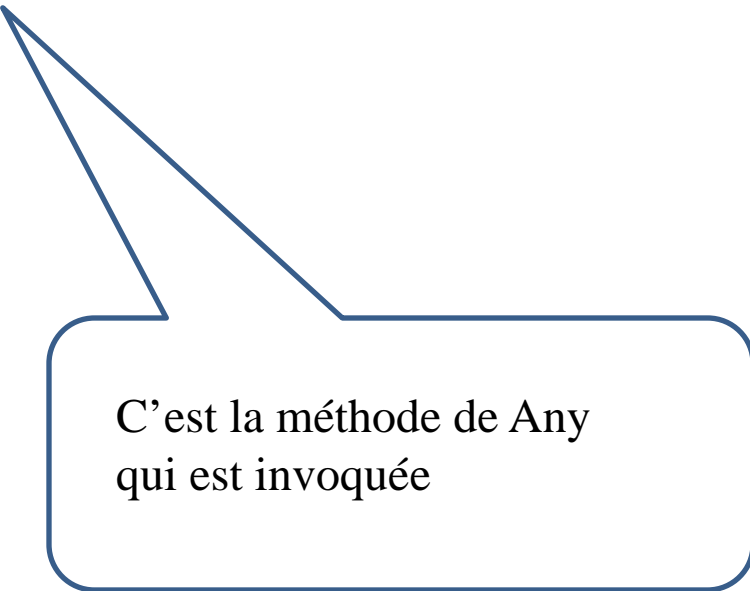
Pourquoi ?



Compléments

```
class Essai() {} ⇔ class Essai():Any {}
```

```
val e=Essai()  
print(e.toString())
```

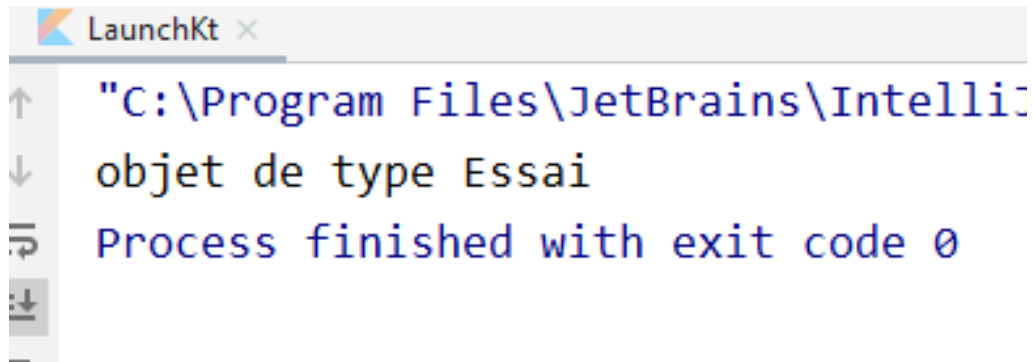


C'est la méthode de Any
qui est invoquée

Compléments

```
class Essai():Any {  
    override fun toString():String=  
        "objet de type Essai"  
}
```

```
val e=Essai()  
print(e.toString())
```



```
LaunchKt x  
↑ "C:\Program Files\JetBrains\IntelliJ  
↓ objet de type Essai  
↻ Process finished with exit code 0  
⏏
```

C'est la méthode de Essai
qui est invoquée

Compléments

```
class Essai {  
    override fun toString():String=  
        "objet de type Essai"  
}
```

```
val e=Essai()  
print(e)
```

Classe abstraite (1)

- Supposons que nous ayons défini deux classes `Carre` et `Cercle` qui chacune définit une méthode `perimetre()`
- On souhaite rendre polymorphe cette méthode : c'est-à-dire pouvoir considérer à la fois des objets de type `Cercle` ou `Carre` et appliquer la méthode `perimetre()`

Classe abstraite (2)

- On veut factoriser la méthode `perimetre()` dans une super classe. On crée la classe `Figure`.
- Cependant, la classe `Figure` ne peut pas mettre en œuvre le calcul du périmètre car ce calcul est différent pour un carré et un cercle.
- La méthode `perimetre()` est dite abstraite :
Seule l'entête est déclarée avec le mot clé `abstract`
La mise en œuvre sera faite par redéfinition dans ses sous classes

Classe abstraite (3)

- Une classe qui contient au moins une méthode abstraite est dite **abstraite**.
- Une classe abstraite n'est pas instantiable car il lui manque des éléments de mise en œuvre de ses méthodes abstraites.