

Développement Orienté Objets

L'héritage en UML et en Kotlin

Arnaud Lanoix Brauer

Arnaud.Lanoix@univ-nantes.fr



Nantes Université

Département informatique

- 1 La notion d'héritage en UML
- 2 L'héritage en Kotlin

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 ?*
- **mutualiser** des parties du code pour éviter la duplication
- mieux **architecturer** le code
- faciliter l'**évolution** du code, la maintenance
- faciliter la **réutilisation** et l'adaptation du code
 - ▶ *polymorphisme*

Relation d'héritage

L'héritage établit une *relation généralisation-spécialisation* $A \leftarrow B$ entre deux classes A et B signifiant que les caractéristiques (attributs et méthodes) de la classe A seront *également présentes* dans la classe B

- La classe A est appelée super-classe, classe "mère", classe de base
- La classe B est appelée sous-classe, classe "fille", classe dérivée
- La classe B *spécialise* la classe A
- La classe A *généralise* la classe B
- La classe B peut avoir des caractéristiques *spécifiques*



Plus généralement, la relation d'héritage \leftarrow est établie entre *une super-classe* et *des sous-classes*

Relation d'héritage

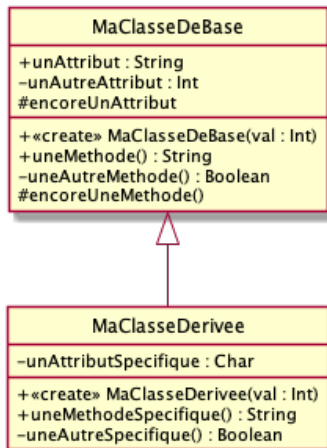
L'héritage établit une *relation généralisation-spécialisation* $A \leftarrow B$ entre deux classes A et B signifiant que les caractéristiques (attributs et méthodes) de la classe A seront *également présentes* dans la classe B

- La classe A est appelée super-classe, classe "mère", classe de base
- La classe B est appelée sous-classe, classe "fille", classe dérivée
- La classe B *spécialise* la classe A
- La classe A *généralise* la classe B
- La classe B peut avoir des caractéristiques *spécifiques*



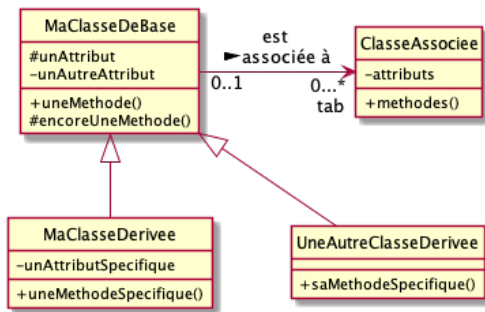
Plus généralement, la relation d'héritage \leftarrow est établie entre *une super-classe* et *des sous-classes*

Décrire une relation d'héritage en UML



- Les attributs de la super-classe **font également partie** de la sous-classe
 - ▶ les attributs **private** ne sont pas accessibles
 - ▶ les attributs **protected** sont uniquement accessibles par les sous-classes
- Les méthodes **public** et **protected** de la super-classe sont appelables **depuis la** sous-classe
 - ▶ Les méthodes **public** de la super-classe sont appelables par les **instances de** la sous-classe
- Les attributs et les méthodes de la sous-classe **ne sont pas accessibles** depuis la super-classe

Décrire une relation d'héritage en UML > les associations

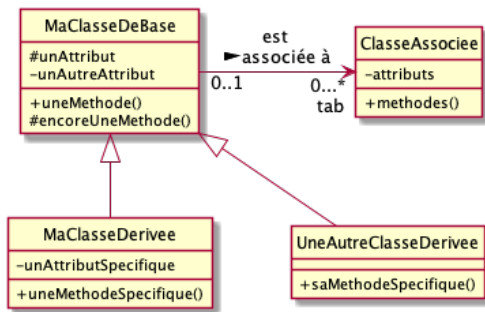


- Plusieurs sous-classes peuvent hériter d'une même super-classe
- Les classes associées à la super-classe **sont également associées** avec les sous-classes
- = une instance de la classe associée peut être en relation avec une instance de la super-classe ou avec une instance d'une sous-classe
- La relation d'héritage **n'a jamais** de nom, ni de cardinalités, ni de rôles

Attention à la notation pour l'héritage

- $B \rightarrow A$ signifie que B "spécialise" A
- $B \rightarrow A$ signifie que B "est associé à" A

Décrire une relation d'héritage en UML > les associations

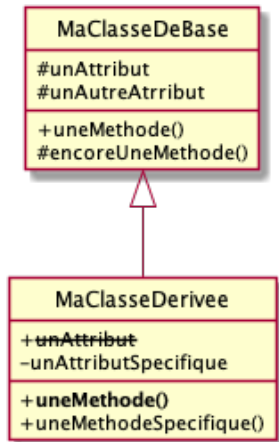


- Plusieurs sous-classes peuvent hériter d'une même super-classe
- Les classes associées à la super-classe **sont également associées** avec les sous-classes
- = une instance de la classe associée peut être en relation avec une instance de la super-classe ou avec une instance d'une sous-classe
- La relation d'héritage **n'a jamais** de nom, ni de cardinalités, ni de rôles

Attention à la notation pour l'héritage

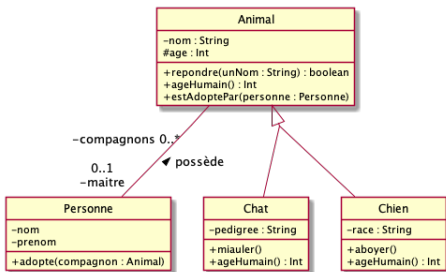
- $B \rightarrow A$ signifie que B "spécialise" A
- $B \rightarrow A$ signifie que B "est associé à" A

Héritage et redéfinition



- On **ne redéclare jamais** les attributs dans la sous-classe
 - ▶ ils sont déjà présents (par héritage)
- On **peut redéclarer** des méthodes dans la sous-classe :
=> **polymorphisme**, c-à-d **redéfinition** de l'implémentation de la méthode
 - ▶ s'il n'a pas de redéfinition, c'est la méthode de la super-classe qui sera utilisée

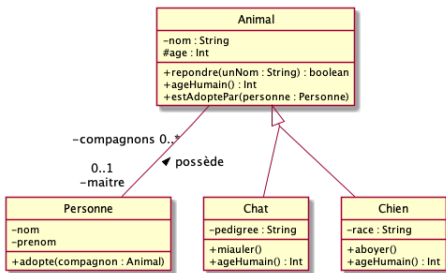
Exemple d'héritage : Chien et Chat



- Chiens et chats sont des animaux
 - ▶ chiens comme chats ont un nom et un âge
 - ▶ répondent à l'appel de leur nom
 - ▶ ont un maitre(sse)
- seul les chiens aboient
- seul les chats miaulent
- Une personne peut posséder plusieurs animaux, qui peuvent être des chiens ou des chats

- Le calcul de l'âge "humain" est différent pour un chien ou pour un chat
- que signifie calculer l'âge humain d'un animal quelconque ?
peut on manipuler un animal quelconque ?

Exemple d'héritage : Chien et Chat



- Chiens et chats sont des animaux
 - ▶ chiens comme chats ont un nom et un âge
 - ▶ répondent à l'appel de leur nom
 - ▶ ont un maitre(sse)
- seul les chiens aboient
- seul les chats miaulent
- Une personne peut posséder plusieurs animaux, qui peuvent être des chiens ou des chats

- Le calcul de l'âge "humain" est différent pour un chien ou pour un chat
- que signifie calculer l'âge humain d'un animal quelconque ?
peut on manipuler un animal quelconque ?

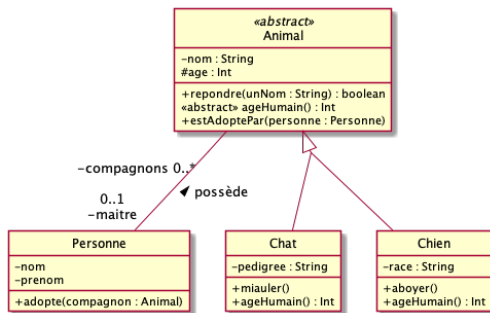
Classe et méthodes abstraites

Une classe **abstraite** ne peut pas être instanciée

- Elle peut déclarer des attributs
- Elle peut contenir des méthodes (concrètes)
- Elle peut déclarer des méthodes **abstraites**, qui
 - ▶ n'auront pas d'implémentation dans la classe `Animal`
 - ▶ devront **obligatoirement** être redéfinies par les sous-classes (concrètes)

On notera une classe / méthode abstraite avec le nom en **italique** ou précédé du stéréotype `<<abstract>>`

Exemple de classe abstraite : la classe Animal



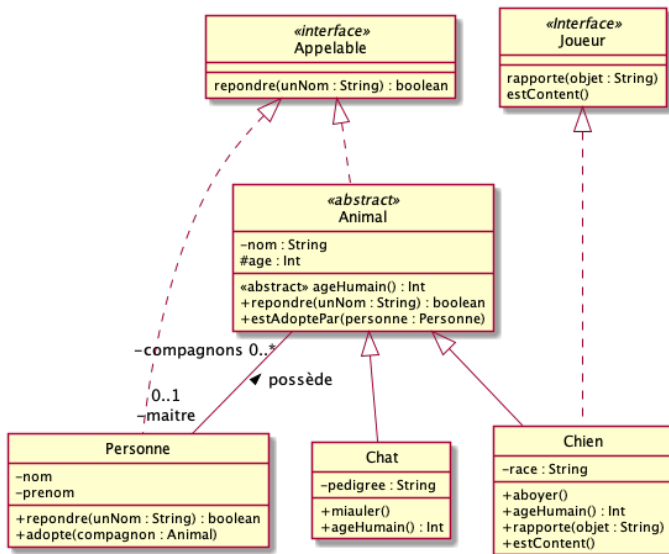
- La classe **Animal** doit être **abstraite** : on ne pourra plus manipuler des instances de **Animal**
- La méthode `ageHumain()` doit être **abstraite** : elle sera redéfinie dans les classes **Chien** et **Chat**

Un personne possède des animaux qui sont, soit des chiens, soit des chats
=> **covariance**

Interface

- Une **interface** est une classe "**complètement**" **abstraite** qui ne déclare que des méthodes abstraites
- Une interface définit un **contrat** =
 - ▶ les classes qui **réalisent** l'interface **doivent** implémenter toutes les méthodes de l'interface
- En UML, une interface se distingue par le stéréotype <<**interface**>>
- La relation de réalisation se distingue de la relation d'héritage

Exemple d'interfaces : Appelable et Joueur



- 1 La notion d'héritage en UML
- 2 L'héritage en Kotlin

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** **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

Utiliser des classes héritées

```
var rogue = Chien("Rogue", 15, "Berger Australien")
var potter = Chien("Potters", 40, "Beauceron")
var gaga = Chat("Gaga", 88, "de maison")
rogue.aboyer()
potter.aboyer()
// potter.miauler() : unresolved reference: miauler
gaga.miauler()
rogue.repondre("Potter")
rogue.repondre("Rogue")
```

- On instancie des objets via le constructeur de la sous-classe
 - ▶ Le constructeur de la super-classe est implicitement appelé
- Les méthodes "spécifiques" à une sous-classe ne sont appelables que sur les instances de cette sous-classe
- Les méthodes de la super-classe sont appelables sur des instances des sous-classes

La covariance en Kotlin

Covariance

La **covariance** consiste à **déclarer** un objet avec le type d'une **super-classe**, puis à **instancier** cet objet avec une **sous-classe**.

- **Restriction** : L'objet a le type de la super-classe ; les méthodes **spécifiques** à la sous-classe ne sont plus accessibles
- **Intérêt** : pouvoir manipuler **indifféremment** des objets ayant le type réel de la classe ou d'une de ses sous-classes
 - ▶ tableaux, attributs d'une autre classe ; paramètres de méthodes, ...

```
var animal : Animal = Chien("Rogue", 16, "Berger Australien")
animal.repondre("Potter")
// animal.aboyer() : unresolved reference: aboyer
animal = Chat("Gaga", 89, "de maison")
animal.repondre("Potter")
```

Interêt de la covariance : exemple

```
class Personne (prenom : String, nom : String) {  
    private val nom : String  
    private val prenom : String  
    private val compagnons : Array<Animal?>  
    private var nbCompagnons : Int  
  
    init {  
        this.nom = nom  
        this.prenom = prenom  
        this.compagnons = arrayOfNulls<Animal>(10)  
        this.nbCompagnons = 0  
    }  
  
    fun adopte(compagnon : Animal) {  
        if (nbCompagnons < compagnons.size) {  
            compagnons[nbCompagnons] = compagnon  
            nbCompagnons++  
        }  
    }  
}
```

```
var rogue = Chien("Rogue", 15, "Berger Australien")  
var gaga = Chat("Gaga", 88, "de maison")  
val arnaud = Personne("Arnaud", "Lanoix Brauer")  
arnaud.adopte(rogue)  
arnaud.adopte(gaga)
```

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)

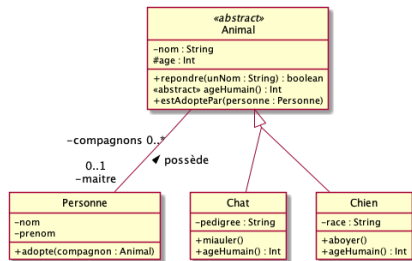
    fun estAdoptePar(p : Personne) {
        maitre = p
    }

    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** des 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 raporte(objet : String)  
  
    fun estContent() {  
        println(" :-) ")  
    }  
}
```

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

