

L'Héritage

Qu'est-ce que l'héritage ?

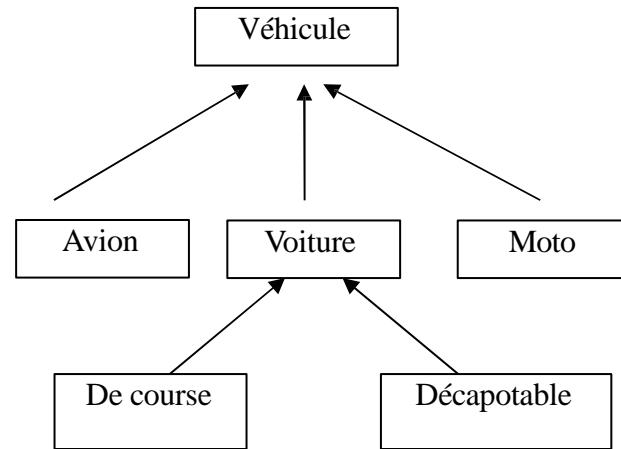
- L'héritage en Java est un concept clé de la programmation orientée objet :

➡ permet de créer de nouvelles classes à partir de classes existantes ;

➡ favorise la réutilisation du code ;

➡ améliore l'organisation et simplifie la maintenance.

L'héritage est mis en œuvre par la construction de classes dérivées. Un exemple de la relation d'héritage est explicité par le graphe suivant :



Hiérarchisation

La classe dont on dérive (qu'on hérite) est dite **CLASSE DE BASE** . Dans

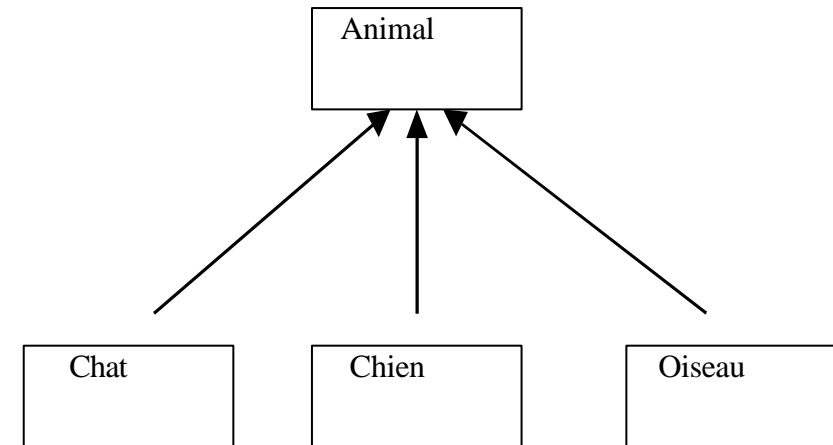
l'exemple suivant :

Animal est la classe de base (classe supérieure),

- les classes obtenues par dérivation sont dites

CLASSES DÉRIVÉES : Chat, Chien et

Oiseau sont des classes dérivées (classes filles).



Classe dérivée (classe fille)

- Une classe dérivée modélise un **cas particulier de la classe de base**, et est **enrichie d'informations supplémentaires**.
- La classe dérivée possède les propriétés suivantes :
 - contient les données membres de la classe de base,
 - peut en posséder de nouvelles,
 - possède (à priori) les méthodes de sa classe de base,

- peut redéfinir (masquer) certaines méthodes,
- peut posséder de nouvelles méthodes.

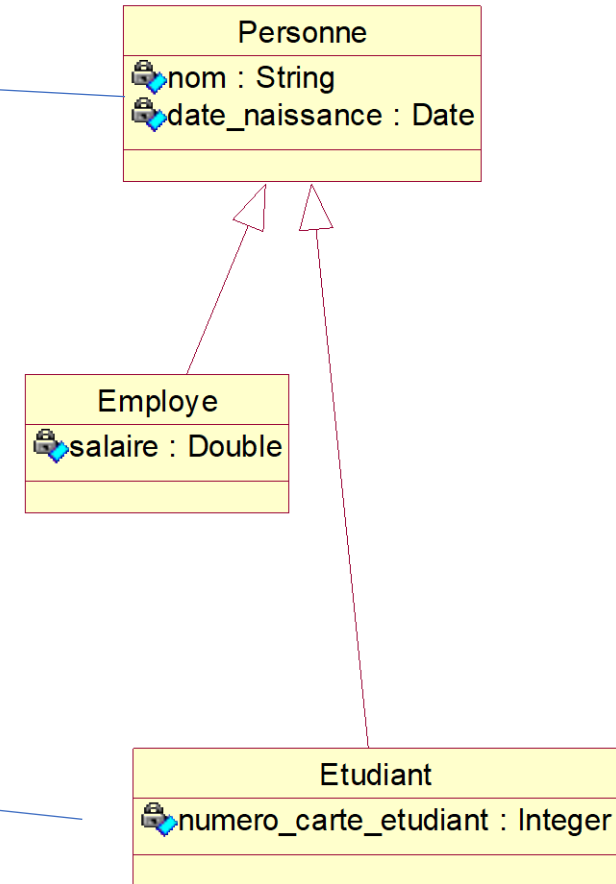
Principe important lié à la notion d'héritage

- Si « B extends A », le grand principe est que **tout B « est un » A**
- Par exemple, un rectangle coloré *est un* rectangle ; une voiture *est un* véhicule
- Ne pas utiliser l'héritage pour réutiliser du code si le principe « est-un » n'est pas vérifié !

```
class Personne
{
    private String nom;
    private Date
date_naissance;
    // ...
}

class Employe extends Personne
{
    private float salaire;
    // ...
}

class Etudiant extends Personne
{
    private int
numero_carte_etudiant;
    // ...
}
```



Utilisation (Syntaxe)

Protection **class** nom de la classe dérivée **extends** nom de la classe
de base {

}

Constructeurs et héritage

- Par défaut le constructeur d'une classe dérivée appelle le constructeur "par défaut" (celui qui ne reçoit pas de paramètres) de la classe de base.

On veillera à ce que le constructeur sans paramètre existe toujours dans la la classe de base.

- La première instruction d'un constructeur peut être un appel
 - à un autre constructeur de la classe : `this(...)`
 - à un constructeur de la classe mère : `super(...)`
- Interdit de mettre `this(...)` et `super(...)` ailleurs qu'au début d'un constructeur

Exemple :

```
class ClasseMere {  
    // Attributs et méthodes de la classe mère  
  
    int i;  
  
    // Constructeur de la classe A  
    public A(int x){ i = x;}  
    -----  
}  
  
class ClasseFille extends ClasseMere {  
    // Attributs et méthodes supplémentaires de la classe fille  
    double z;  
  
    // Constructeur de la classe B  
    public B(int f, double w) {  
        // Appel explicite du constructeur de A,  
        // et en 1ère ligne.  
        super(f);  
        z = w;  
    }  
    -----  
}
```

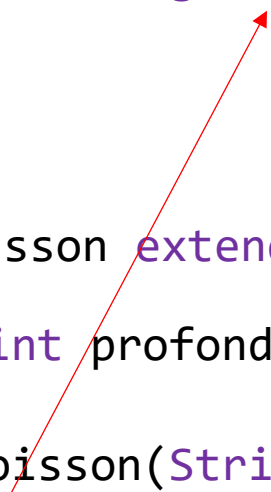
Droits d'accès :

- Si les membres de la classe de base sont :
 - **public** ou « rien » : les membres de la classe dérivée auront accès à ces membres (champs et méthodes),
 - **private** : les membres de la classe dérivée n'auront pas accès aux membres privés de la classe de base.
- Il existe un 4^e niveau de protection : **protected**.

Un membre de la classe de base déclaré **protected** est accessible à ses **classes dérivées** ainsi qu'aux classes du même package.

Exemple d'utilisation de **protected**

```
public class Animal {  
    protected String nom;  
    . . .  
}  
  
public class Poisson extends Animal {  
    private int profondeurMax;  
  
    public Poisson(String unNom, int uneProfondeur) {  
        nom = unNom;    // utilisation de nom  
        profondeurMax = uneProfondeur;  
    }  
}
```



Redéfinition de méthodes

- Une sous-classe peut redéfinir des méthodes existant dans une de ses superclasses (directe ou indirectes), à des fins de spécialisation.
 - Le terme anglophone est "overriding". On parle aussi de masquage.
 - La méthode redéfinie **doit avoir la même signature**.

```
class Employe extends Personne
{
    private float salaire;
    public calculePrime( )
    {
        // ...
    }
}
```

Redéfinition

```
class Cadre extends Employe
{
    @Override
    public calculePrime()
    {
        // ...
    }
    // ...
}
```

Classe Object

- La racine de l'arbre d'héritage des classes est la classe `java.lang.Object`
- Pas de variables d'état (ni d'instance, ni de classe), pas de méthodes `static`
- Les méthodes d'instance de Object sont héritées par toutes les classes
- Les plus utilisées sont les méthodes
 - `toString`, `equals`, `hashCode` et `getClass`

Opérateur `instanceof`

- L'opérateur `instanceof` confère aux instances une capacité d'introspection : il permet de savoir si une instance est instance d'une classe donnée.
 - Renvoie une valeur booléenne

```
if ( ... )  
    Personne jean = new Etudiant();  
else  
    Personne jean = new Employe();  
  
//...
```

```
if (jean instanceof Employe)  
    // discuter affaires  
else  
    // proposer un stage
```


Classes et méthodes **final**

- Classe **final** : ne peut pas avoir de classes filles (**String** est **final**)
- Méthode **final** : ne peut pas être redéfinie
- Variable (locale ou d'état) **final** : la valeur ne peut pas être modifiée après son initialisation
- Paramètre **final** : la valeur ne peut pas être modifiée dans le code de la méthode

Classes et méthodes abstraites

Méthode abstraite : c'est une méthode sans implémentation.
Elle sera implémentée dans les classes filles

```
public abstract int m(String s);
```

La méthode m sera implémentée dans les classes dérivées (filles)

Classe abstraite

Une classe abstraite est une classe dans laquelle certaines méthodes sont seulement déclarées, et pas définies. Une telle classe contient, comme une classe ordinaire : des déclarations d'attributs et des définitions de constructeurs et de méthodes. Elle peut contenir, en plus, des **méthodes abstraites**, c'est-à-dire de simples déclarations de méthodes .

Une déclaration de méthode indique **le type de retour, les types des arguments**, et termine par un point-virgule sans code ni accolades.

Chaque méthode ou classe abstraite est accompagnée du mot-clé **abstract**. Une classe **abstraite ne peut pas être instanciée**

Exemple

```
abstract class Animal {  
    abstract void faireDuBruit();  
  
    void dormir() {  
        System.out.println("Cet animal dort.");  
    }  
}  
  
class Chien extends Animal {  
    void faireDuBruit() {  
        System.out.println("Le chien aboie.");  
    }  
}
```

Test de l'exemple

```
public class Test {  
    public static void main(String[] args) {  
        // Animal animal = new Animal();  
        // Erreur : Impossible d'instancier une classe abstraite  
        Animal animal = new Chien();  
        Animal.faireDuBruit();  
        animal.dormir();  
    }  
}
```

Exemple d'application Complet

I) Classe de base

- On se propose de créer une classe véhicule ayant les attributs suivants :

```
marque  (String)
modele  (String)
annee   (int)
```

- 1) Créer cette classe.
- 2) Ajouter un constructeur prenant ces trois attributs comme paramètres
- 3) Ecrire la méthode `afficherInfos()` qui affiche les informations du véhicule
- 4) Ecrire la méthode `demarrer()` qui affiche : « la voiture démarre »

II-Classes filles

Créer les classes voiture et moto qui héritent de véhicule.

1) Dans la classe Voiture :

a) ajouter un attribut : nombre de portes (`int`)

b) redéfinir la méthode `demarre()` pour afficher

« la voiture démarre en tournant la clé. »

2) Dans la classe Moto :

a) ajouter un attribut `cylindree` (`int`)

b) redéfinir la méthode `demarrer()` pour afficher « la moto démarre en appuyant sur un bouton »

III - Classe TestVehicule

Créer plusieurs véhicules, voitures et motos qui seront stockés dans une variable `garage` de type `ArrayList`

Afficher la liste de tous les véhicules avec leurs informations (caractéristiques et méthode de démarrage)