IIT
INSTITUT INTERNATIONAL TECHNOLOGIE
Université Nord Américaine privée

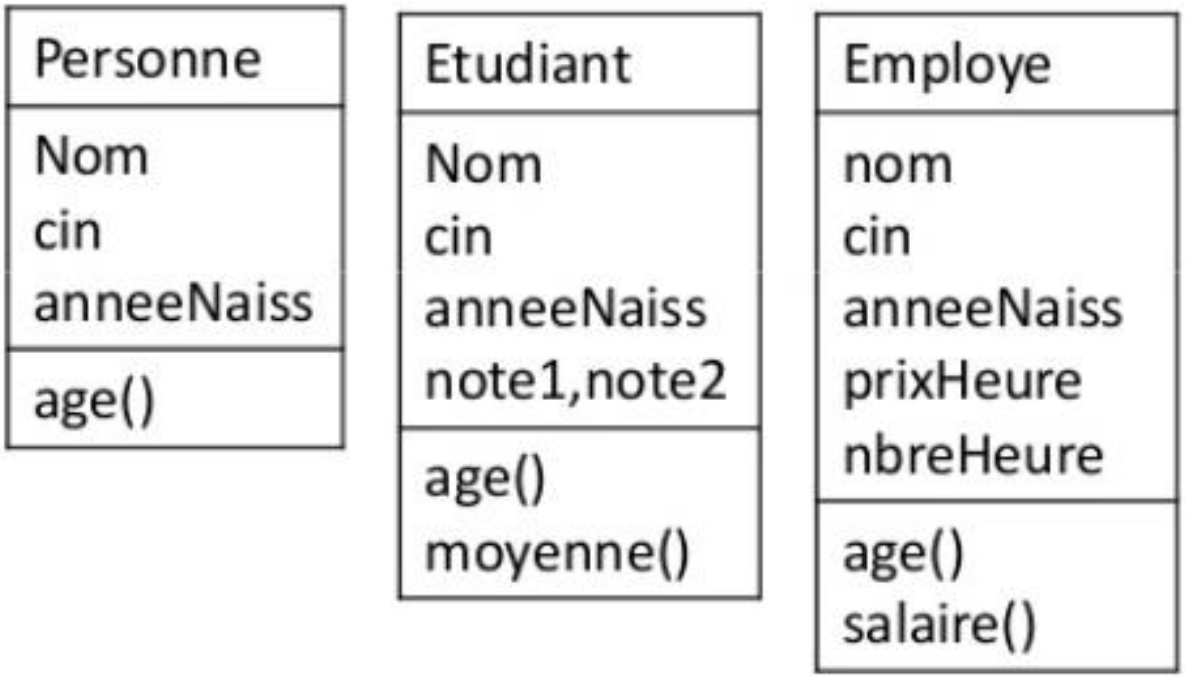
Chapitre 4 : Héritage et Polymorphisme

Enseignante : Fairouz Fakhfakh

fairouz.fakhfakh@iit.ens.tn
fairouz.fakhfakh@gmail.com

Notion d'héritage

- Définition de 3 classes



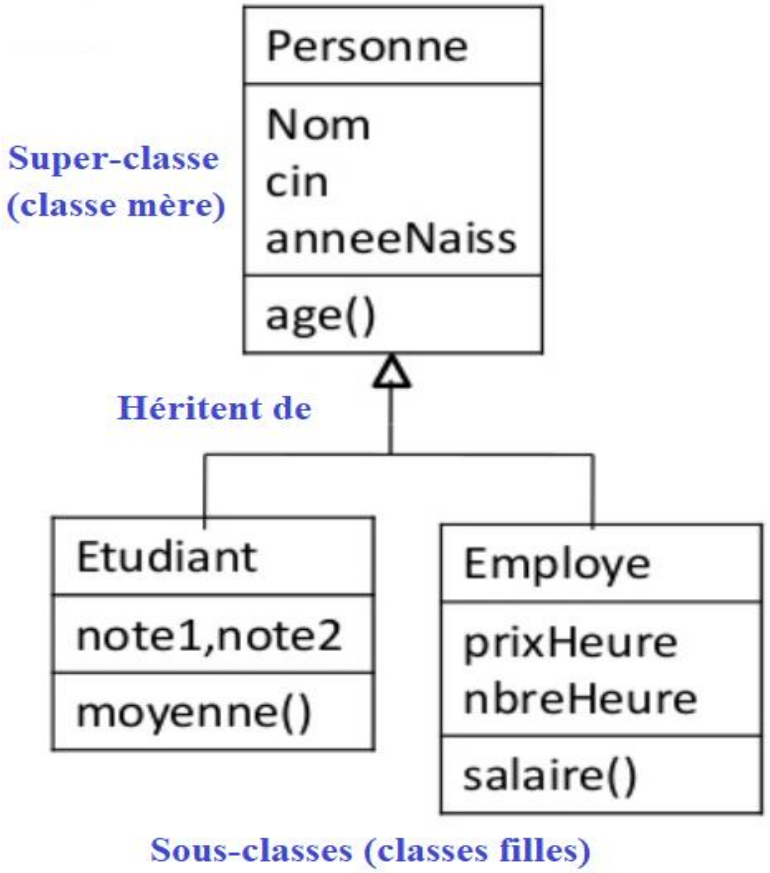
- **Limites**

✓ Duplication du code

✓ Une modification faite sur un attribut (ou méthode) commun doit être refaite dans les autres classes.

Notion d'héritage

- **Solution**



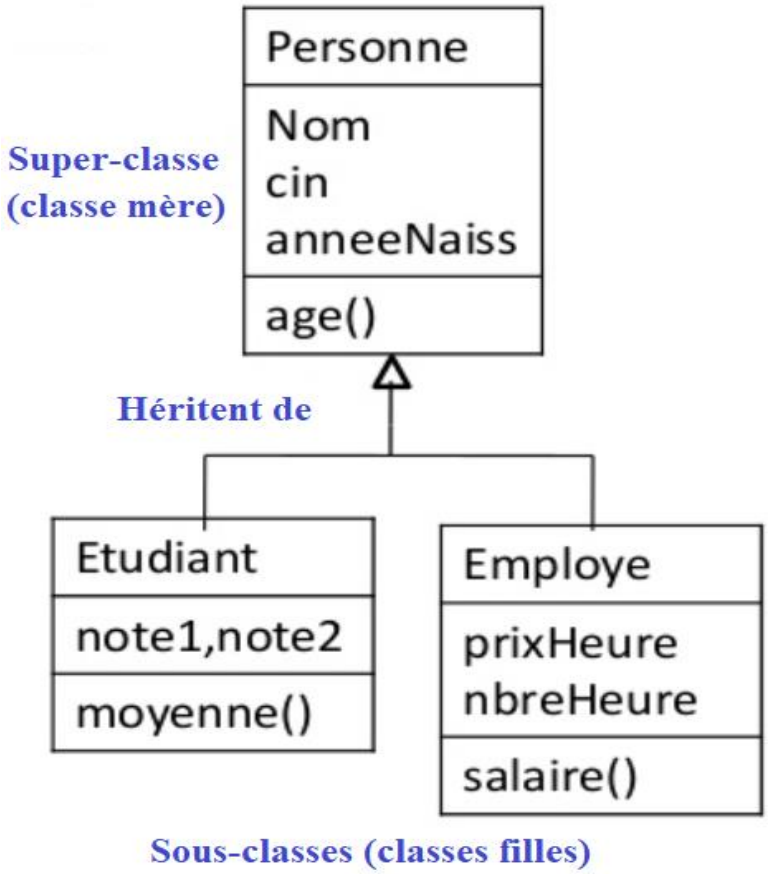
Placer dans la classe mère tous les attributs et toutes les méthodes communs à toutes les classes.

Les classes filles ne comportent que les attributs ou méthodes plus spécifiques.

Les classes filles héritent automatiquement les attributs (et les méthodes) qui n'ont pas besoin d'être ré-écrits.

Notion d'héritage

- **Solution**



- **Avantages**
  - ✓ Réutilisation du code
  - ✓ Une seule modification des attributs (ou méthodes) en commun.

Notion d'héritage

```
class Personne {

String nom;
int cin, anneeNaiss;

int age(){
        return 2020 - anneeNaiss;
          }
```

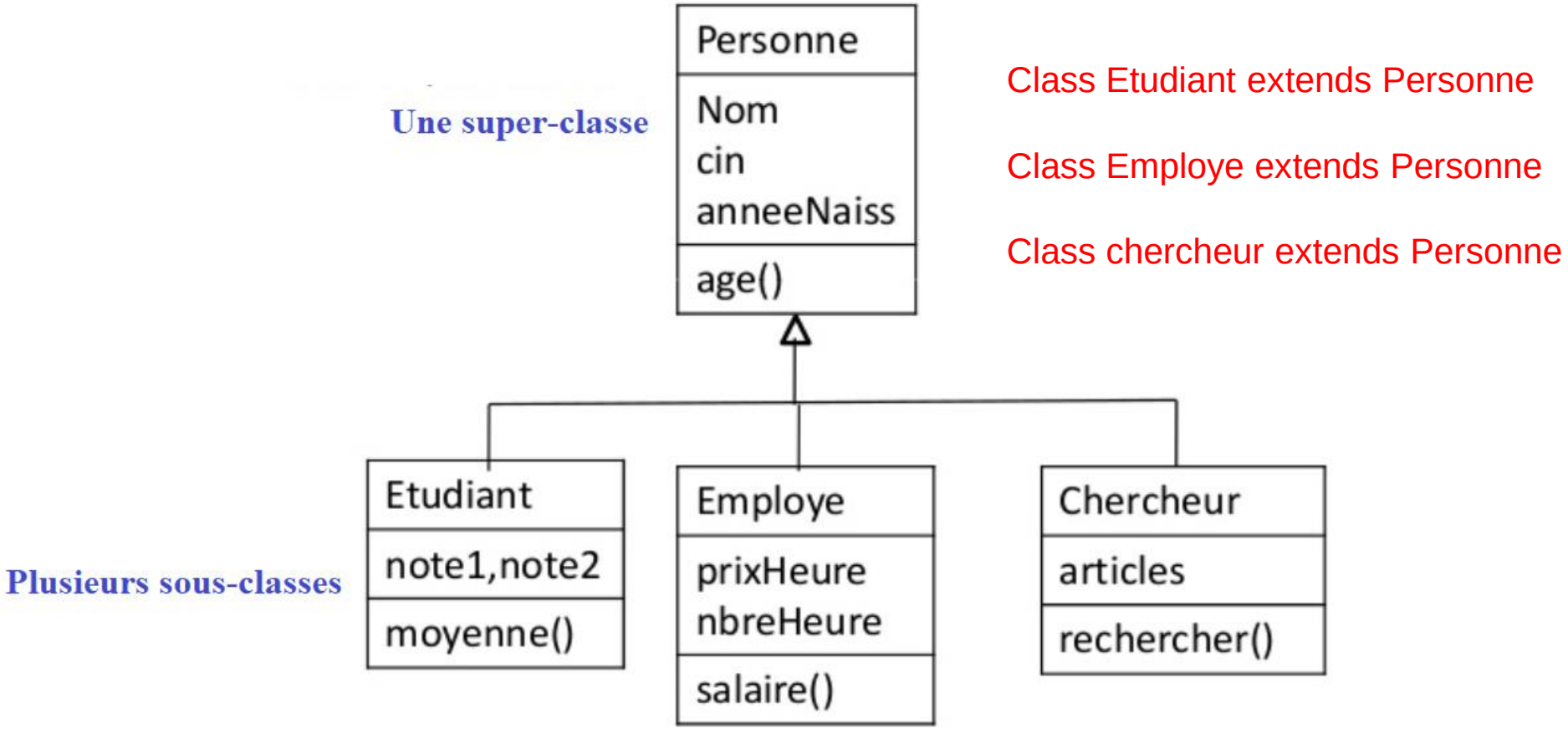
```
class Etudiant extends Personne {

float note1,note2;


public float moyenne (){
        return ((note1+note2)/2);}
}
```

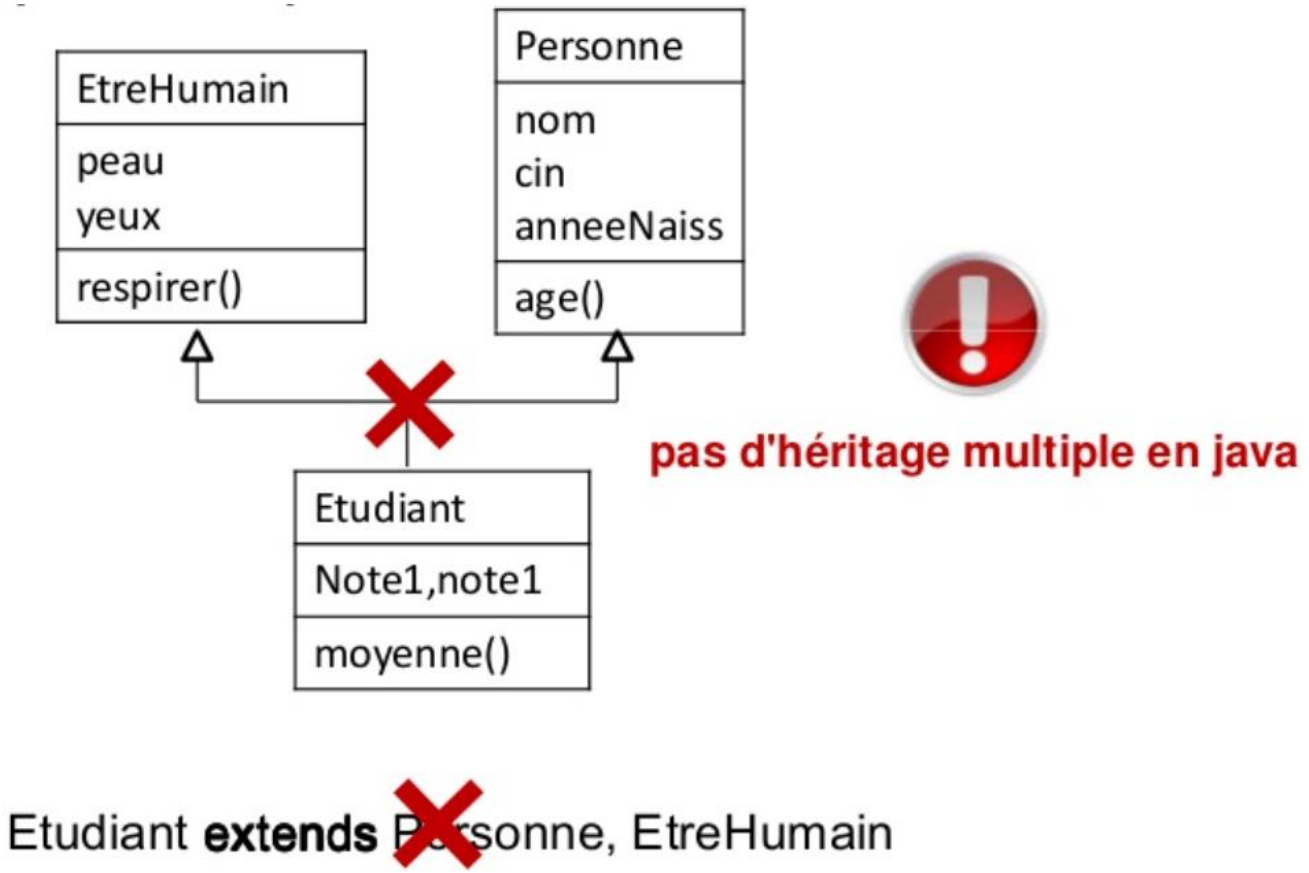
Notion d'héritage

- Une classe peut avoir plusieurs sous-classes.



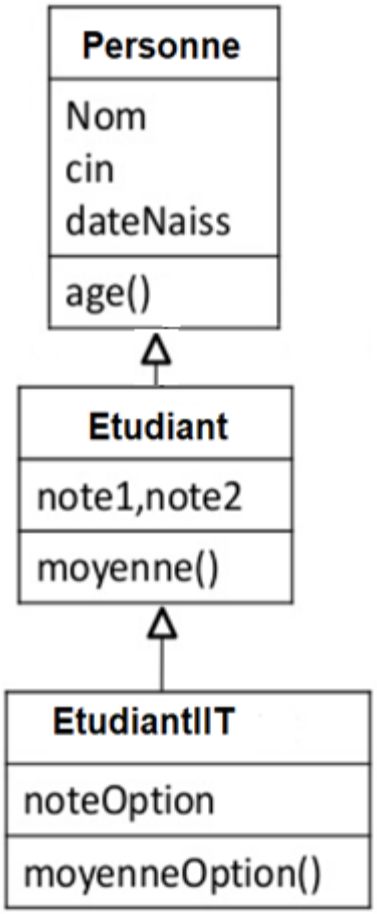
Notion d'héritage

- Une classe ne peut avoir qu'une seule classe mère.



Notion d'héritage

- **Héritage en cascade**



- ✓ Etudiant hérite de Personne
- ✓ EtudiantIIT hérite de Etudiant

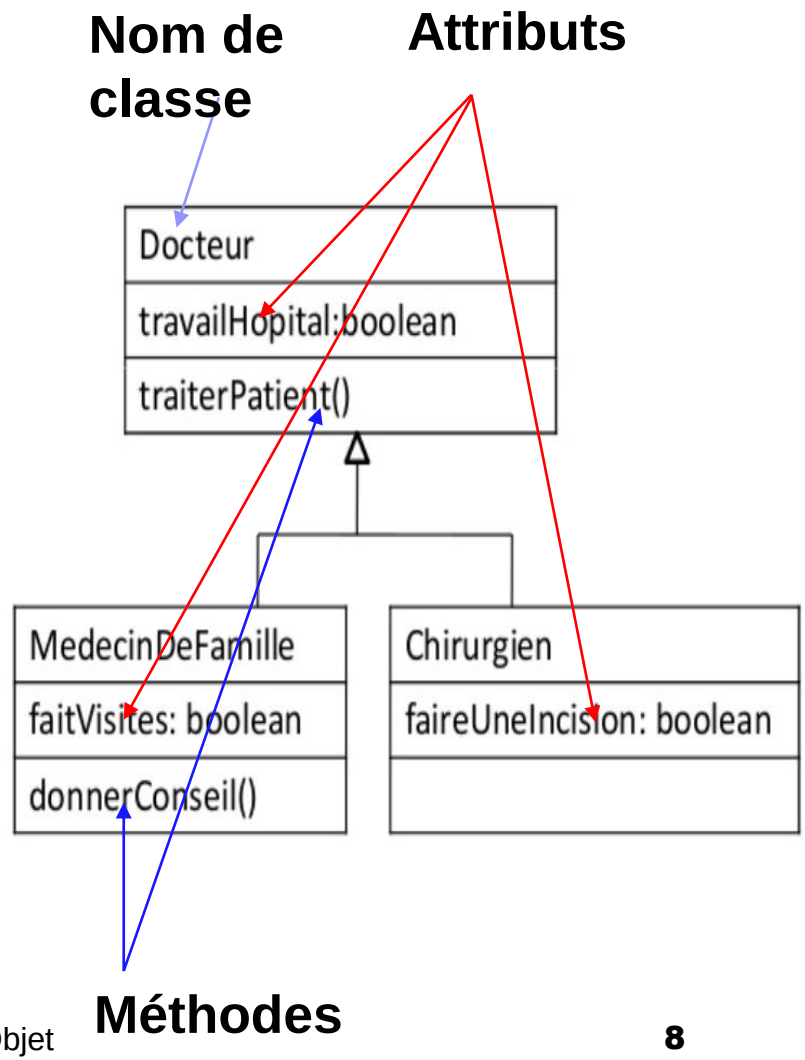
} EtudiantIIT hérite de Personne

```
Etudiant etudiant = new Etudiant ();
etudiant.age();
etudiant.moyenne();
etudiant.moyenneOption();
```

```
EtudiantIIT etudiantiit = new EtudiantIIT ();
etudiantiit.age();
etudiantiit.moyenne();
etudiantiit.moyenneOption();
```

Notion d'héritage

- ✓ Quel est le nombre d'attributs de Chirugien ?
- ✓ Quel est le nombre d'attributs de MdecinDeFamille ?
- ✓ Combien de méthodes Docteur a-t-il ?
- ✓ Combien de méthodes MedecinDeFamille a-t-il ?
- ✓ MdecinDeFamille peut-il invoquer traiterPatient() ?
- ✓ MdecinDeFamille peut-il utiliser faireUneIncision ?



Le mot-clé « super »

Le mot-clé *super* permet de désigner la superclasse,

➡ *super* permet d'accéder aux attributs et méthodes de la super-classe

- Pour manipuler un attribut de la super-classe :

  super.cin

- Pour manipuler une méthode de la super-classe :

  super.age()

- Pour faire appel au constructeur de la super-classe:

  super(nom, cin)

Constructeurs et héritage

- **Constructeur par défaut**

```
class Personne {

private String nom;
private long cin;

public Presonne(){
        nom="Mohamed";
        cin=00000000;
}

}
```

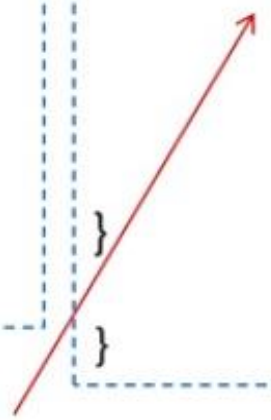
```
class Etudiant extends Personne {

private float note1,note2;

public Etudiant(){
        super();
        note1=0.0f;
        note2=0.0f;

}

}
```



super fait appel au constructeur par défaut de la classe mère

Constructeurs et héritage

- Par défaut le constructeur d'une sous-classe appelle le constructeur "par défaut" (celui qui ne reçoit pas de paramètres) de la superclasse.
- **Attention** : Dans ce cas que le constructeur sans paramètre existe toujours dans la superclasse...
- Pour forcer l'appel d'un constructeur précis, on utilisera le mot réservé **super.** Cet appel devra être la **première instruction** du constructeur.

Constructeurs et héritage

- **Constructeur surchargé**

```
class Personne {

 public  String nom;
 public  long cin;

public Presonne(String nom, long cin){
            this.nom=nom;
            this.cin=cin;
}

}
```

```
class Etudiant extends Personne {

 public  float note1,note2;

public Etudiant(String nom, long cin, int
note1, int note 2){
            super(nom, cin);
            this.note1=note1;
            this.note2=note2;
}

}
```

super fait appel au constructeur surchargé de la classe mère

Constructeurs et héritage

- La **première** instruction dans le constructeur de la sous-classe doit être l'appel au constructeur de la superclasse avec le mot clé **super**.
- Si on ne fait pas l'appel explicite au constructeur de la superclasse, c'est la **constructeur par défaut** de la superclasse qui est appelé implicitement.

```
public class Animal {

   public int nbPattes;

   public Animal(int nbPattes) {
            this.nbPattes = nbPattes; }
   }
```

```
public class Chat extends Animal{

}
```

Constructeurs et héritage

```
public class Animal {

   public int nbPattes;

   public Animal(int nbPattes) {
            this.nbPattes = nbPattes; }
   }
```

```
public class Chat extends Animal{

   public Chat(){
            super();
    }
}
```

Constructeur par défaut
Appel implicite à super

- Un constructeur surchargé est créé par la classe Animal
  => Le constructeur par défaut n'existe pas.
- Erreur à la compilation lors de l'exécution de super().

Constructeurs et héritage

- **Solution 1 :** Déclarer explicitement le constructeur par défaut de la classe mère

```
public class Animal {

   public int nbPattes;

   public Animal(){}

   public Animal(int nbPattes) {
            this.nbPattes = nbPattes; }
   }
```

```
public class Chat extends Animal{

}
```

Constructeurs et héritage

- **Solution 2 :** Faire un appel explicite au constructeur surchargé de la classe mère

```
public class Animal {

   public int nbPattes;

   public Animal(int nbPattes) {
            this.nbPattes = nbPattes;
    }
}
```

```
public class Chat extends Animal{

  public Chat(){
            super(4);
  }
}
```

Redéfinition des méthodes

```
class Personne {

    public void lire() {
            System.out.println("Je sais lire");
    }

}
```

```
class Etudiant extends Personne {

 public void lire() {
            System.out.println("Je suis un étudiant");
            System.out.println("Je sais lire");
    }
}
```

- La classe Etudiant hérite la méthode lire() mais on veut changer son traitement

- On garde **la même signature** de la méthode lire et on change le corps

↳ On parle alors de **redéfinition (override)**

Redéfinition des méthodes

```
class Personne {

    public void lire() {
            System.out.println("Je sais lire");
    }

}
```

```
class Etudiant extends Personne {

 public void lire() {
            System.out.println("Je suis un etudiant");
            super.lire();
    }
}
```

➡ Possibilité de réutiliser le code de la méthode héritée (super)

Surcharge des méthodes

```
class Personne {

    public void lire() {
            System.out.println("Je sais lire");
    }

}
```

```
class Etudiant extends Personne {

    public void lire(String titreLivre) {
            System.out.println("Je sais lire");
            System.out.println("J'ai lu le livre "+titreLivre);
    }
}
```

- On garde le même nom de la méthode, le même type de retour et on change les paramètres

On parle alors de **surcharge (overload)**

Surcharge VS Redéfinition

```
class Personne {

    public void lire() {
            System.out.println("Je sais lire");
    }
}
```

surcharge

```
class Etudiant extends Personne {

 public void lire(String titreLivre) {
            System.out.println("Je sais lire");
            System.out.println("J'ai lu le livre "+titreLivre);
    }
}
```

Etudiant possède deux méthodes

- `lire()`
- `lire(String)`

redéfinition

```
class Etudiant extends Personne {

 public void lire() {
            System.out.println("Je suis un étudiant");
            super.lire();
    }
}
```

Etudiant possède une seule méthode

- `lire()`

Surcharge VS Redéfinition

- **Remarque :**

Ne pas confondre "redéfinition" et "surcharge". Si une classe fille déclare une méthode ayant le même nom que celui d'une méthode existante dans une classe parente, il y a deux possibilités :

- les paramètres et le type retourné sont les mêmes : il s'agit bien d'une « **redéfinition de méthode** »
- les paramètres de la méthode ne sont plus les mêmes. Il s'agit alors d'une « **surcharge de méthode** » et non pas d'une redéfinition.

Le mot clé final

- **Empêcher la redéfinition**
  - Une classe peut protéger une méthode afin d'éviter qu'elle ne soit redéfinie dans ses sous-classes.
  - En Java, on va simplement ajouter le modificateur **final** dans la signature de la méthode.

```
public final void lire()
```

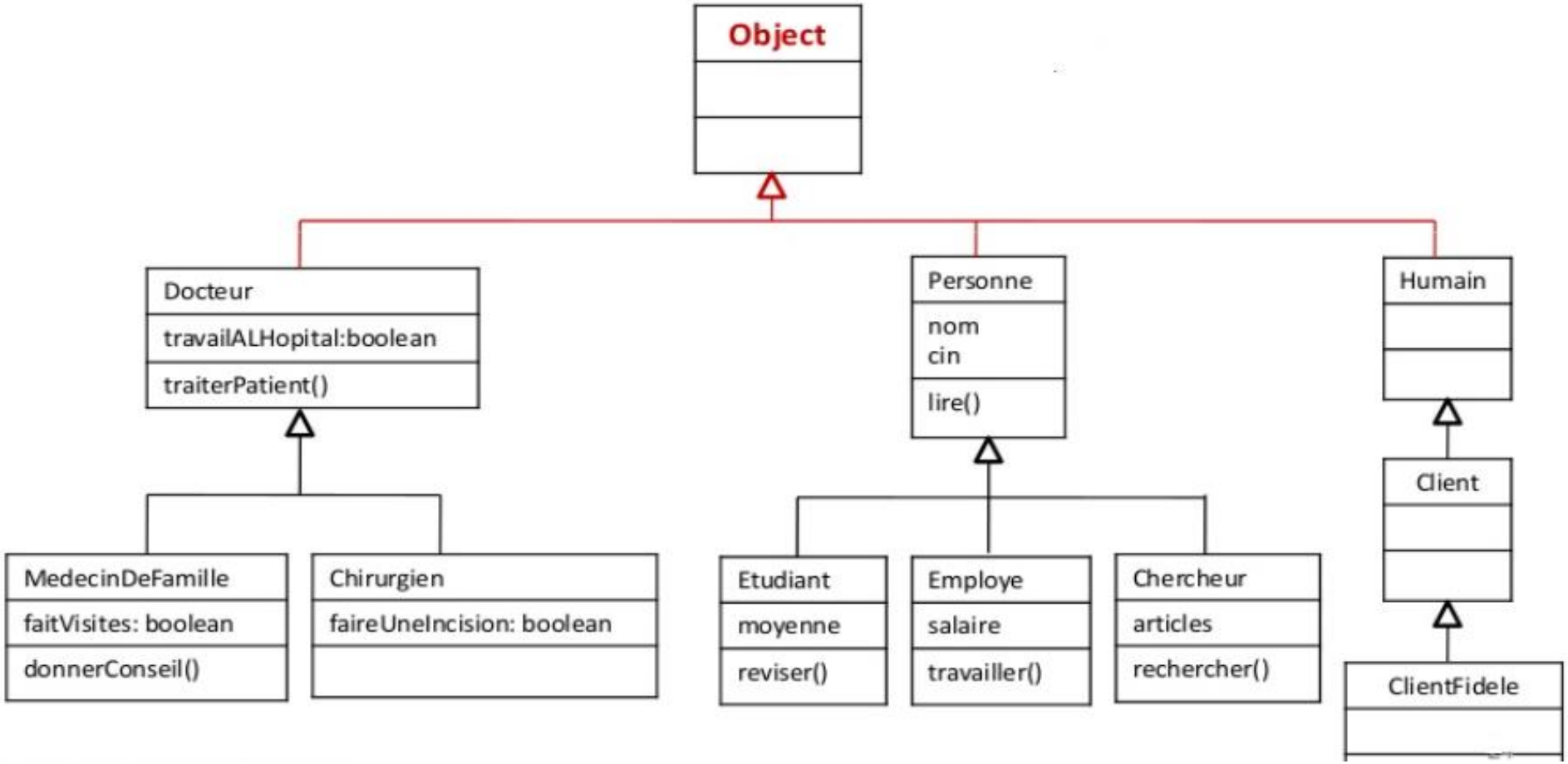
- **Empêcher l'héritage**
  - Une classe déclarée final ne peut pas être dérivée

```
final class Personne
```

  - Permet d'interdire tout héritage pour cette classe qui ne pourra pas être une classe mère
  - Toutes les méthodes à l'intérieur de la classe final seront implicitement finales.

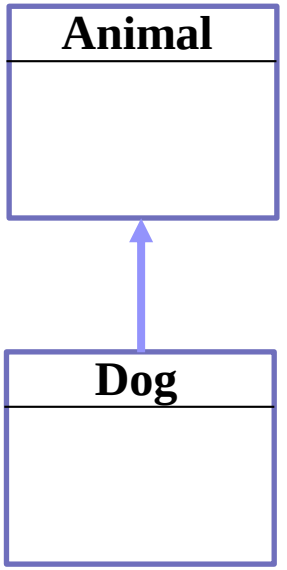
La classe Object

- Toutes les classes Java héritent **implicitement** de la classe **Object**.
- La classe Object est l'ancètre de toutes les classes.



Polymorphisme

- N'importe quel objet en Java qui peut passer d'une relation « est un » peut être considéré polymorphe.
- Le polymorphisme est le fait de référencer une classe ***fille*** avec un référence déclaré de type une classe ***mère***.




```
Animal myDog = new Dog();
```

Type déclaré ou type de référence = type de la variable

Référence déclaré en tant qu'Animal().

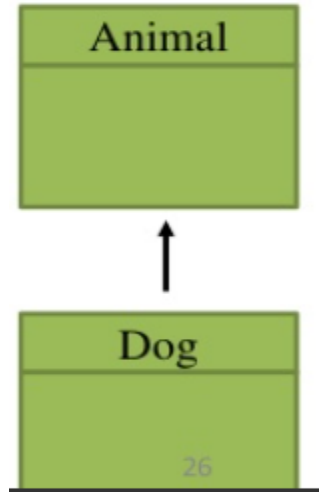
Objet crée en tant que Dog().

Type de l'objet ou Type courant

Le type déclaré définit l'ensemble des opérations qui peuvent être accomplies sur la variable, Le compilateur vérifie que cette règle est respectée.

Polymorphisme

- N'importe quel objet en Java qui peut passer d'une relation « est un » peut être considéré polymorphe.
- Le polymorphisme est le fait de référencer une classe ***fille*** avec un référence déclaré de type une classe ***mère***.



Animal myDog = new Dog();

Type déclaré ou type de référence = type de la variable

Référence déclaré en tant qu'Animal().

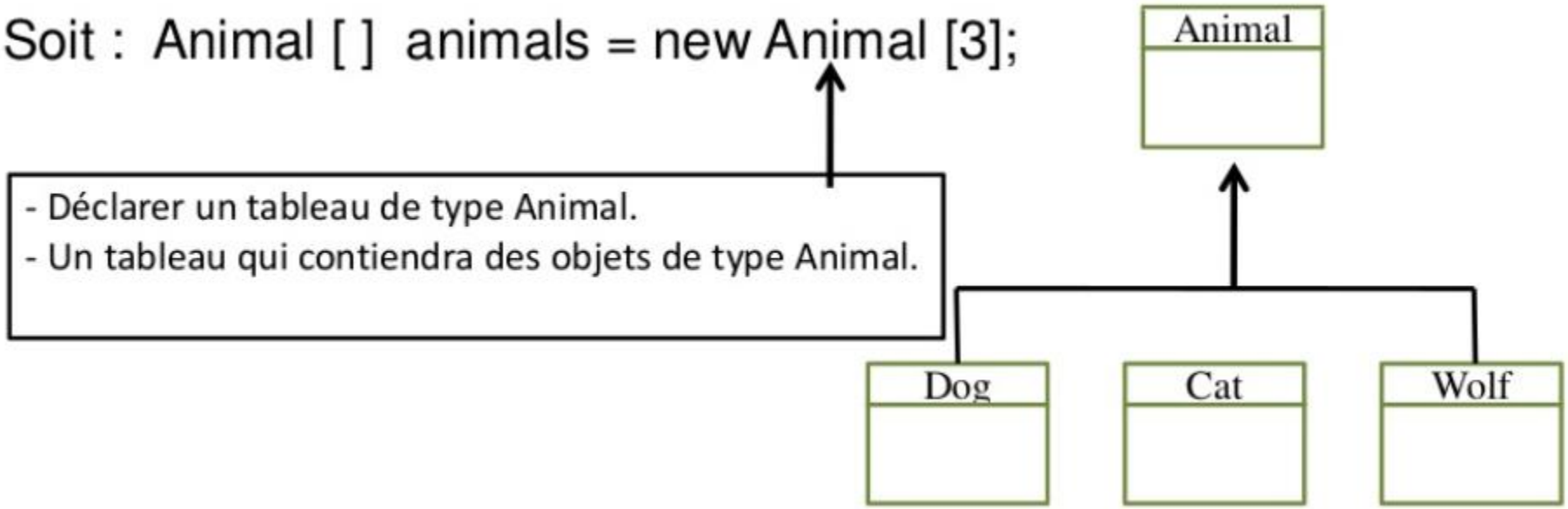
Objet crée en tant que Dog().

Type de l'objet ou Type courant

Type courant ou type réel = type de l'objet
Le type courant est dynamique, connu uniquement à l'exécution

Polymorphisme

- Avec le polymorphisme : le type de la référence peut être la classe mère de l'objet instancié.



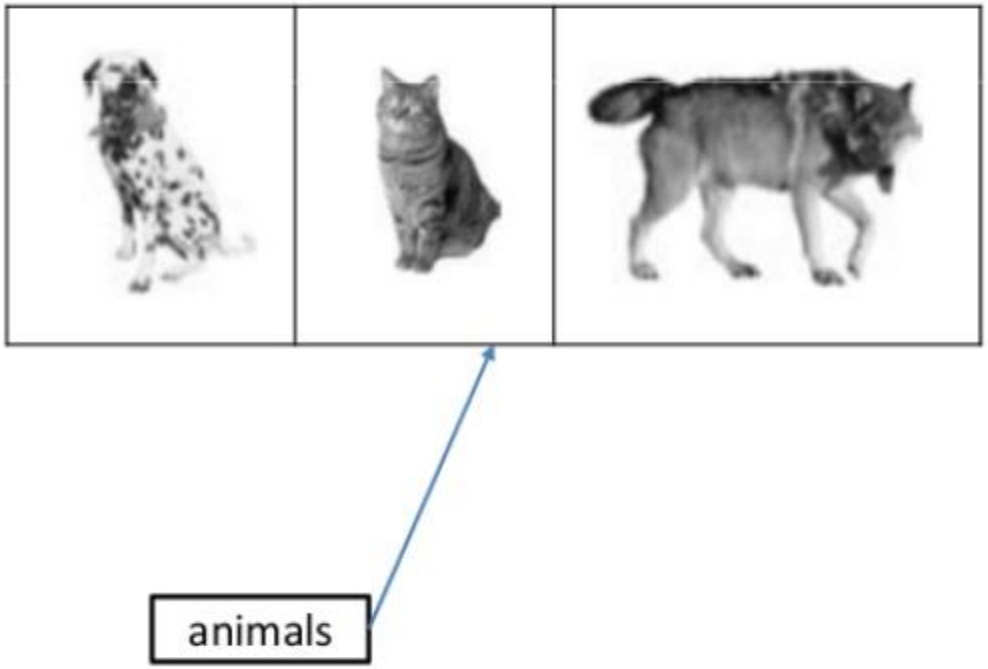
Polymorphisme

- On peut mettre n'importe quelle classe fille de la classe Animal() dans le tableau.

```
animals [0] = new Dog();

animals [1] = new Cat();

animals [2] = new Wolf();
```



Polymorphisme

Les méthodes polymorphes

- Une méthode polymorphe est une méthode déclarée dans une super classe et redéfinie par une sous classe.
- Les méthodes polymorphes permettent de prévoir des opérations similaires sur des objets de nature différentes.

Polymorphisme : Exemple 1

```java
public class Animal {

    public void eat(){
        System.out.println("Un animal peut manger !!");
    }
    public void roam(){
        System.out.println("Un animal peut voyager !!");
    }

}
```

```java
public class Cat extends Animal {

    @Override
    public void eat() {
        System.out.println("Je suis un CHAT, Je mange");
    }

    @Override
    public void roam() {
        System.out.println("Je suis un CHAT, Je voyage");
    }
}
```

```java
public class Dog extends Animal {

    @Override
    public void eat() {
        System.out.println("Je suis un CHIEN, Je mange");
    }

    @Override
    public void roam() {
        System.out.println("Je suis un CHIEN, Je voyage");
    }
}
```

```java
public class Wolf extends Animal {

    @Override
    public void eat() {
        System.out.println("Je suis un LOUP, Je mange");
    }

    @Override
    public void roam() {
        System.out.println("Je suis un LOUP, Je voyage");
    }
}
```

Polymorphisme : Exemple 1

➔ On va parcourir le tableau et appeler n'importe quelle méthode de manière ***concrète***.

```
**********
Je suis un CHAT, Je mange
Je suis un CHAT, Je voyage
**********
**********
Je suis un CHIEN, Je mange
Je suis un CHIEN, Je voyage
**********
**********
Je suis un LOUP, Je mange
Je suis un LOUP, Je voyage
**********
```

```
public class Test {

    public static void main(String[] args) {
        Animal[] animals = new Animal[3];
        animals[0] = new Cat();
        animals[1] = new Dog();
        animals[2] = new Wolf();

        for (int i = 0; i < animals.length; i++) {
            System.out.println("**********");
            animals[i].eat();
            animals[i].roam();
            System.out.println("**********");
        }

    }

}
```

Polymorphisme : Exemple 2

```
class Personne {
public void affiche() {System.out.println("Personne");}
}
class Etudiant extends Personne {
public void affiche() {System.out.println("Etudiant");}
}
class Employe extends Personne {
public void affiche() {System.out.println("Employe");}
}

class Population {
public static void main (String[] args) {
Personne pop[]= new Personne[3];
pop[0]= new Etudiant();
pop[1]= new Personne();
pop[2]= new Employe();
for(int i=0; i<3; i++)
    pop[i].affiche();
}}
```

Polymorphisme

- **Quelle est la méthode appelée ?**

La méthode appelée est celle existante dans la classe du **type le plus *inférieur* dans l'hiérarchie d'héritage gagne !**

La machine virtuelle *JVM* commence tout d'abord à voir dans la classe *Wolf* .
Si elle ne trouve pas une correspondance de la version de la méthode, elle commence à grimper l'hiérarchie de l'héritage jusqu'à trouver la bonne méthode.

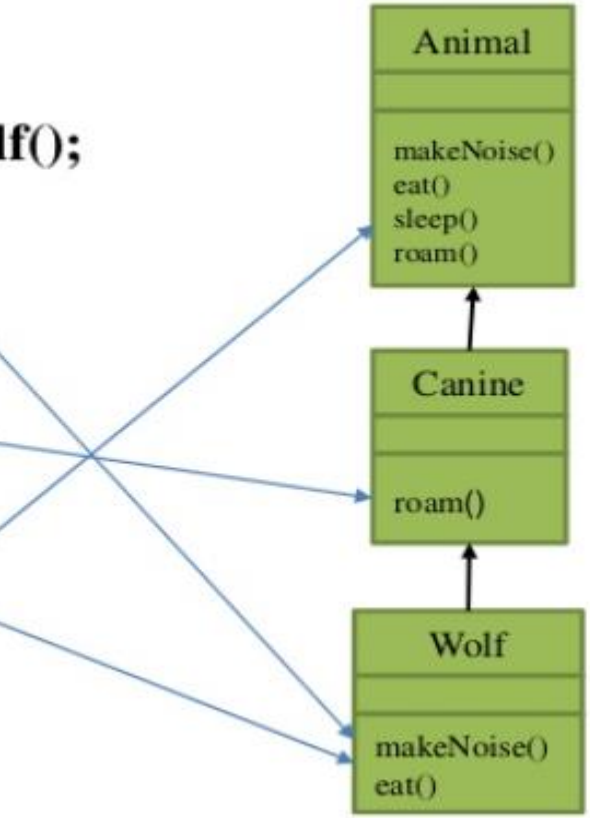
Wolf w = new **Wolf();**

w.makeNoise();

w.roam();

w.eat();

w.sleep();



Forçage de type/ transtypage

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

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

Personne jean = new Employe ();
float i = jean.salaire;  // Erreur de compilation
float j = ( (Employe) jean ).salaire;    // OK
```

A ce niveau pour le Compilateur, la variable « jean » est un objet de la classe Personne, donc il n'a pas d'attribut «salaire»

On « force » le type de la variable « jean » pour pouvoir accéder à l'attribut «salaire». On peut le faire car c'est bien un objet Employe qui est dans cette variable.

Forçage de type/ transtypage

- Lorsqu'une référence de type d'une classe désigne une instance d'une sous-classe, il est nécessaire de **forcer le type de la référence** pour accéder aux attributs spécifiques à la sous-classe.
- Si ce n'est pas fait, le compilateur ne peut pas déterminer le type réel de l'instance, ce qui provoque une erreur de compilation.
- On utilise également le terme de transtypage similaire au « cast » en C,

L’autoréférence : this

- Le mot réservé **this**, utilisé dans une méthode, désigne la référence de l'instance à laquelle le message a été envoyée (donc celle sur laquelle la méthode est « exécutée »).
- Il est utilisé principalement :
  - lorsqu'une référence à l'instance courante doit être passée en paramètre à une méthode,
  - pour lever une ambiguïté,
  - dans un constructeur, pour appeler un autre constructeur de la même classe.

L'autoréférence : this

```
class Personne
  {
     public String nom;
     Personne (String nom)
       {
          this.nom=nom;
       }
  }
```

Pour lever l'ambiguïté sur le mot « nom » et déterminer si c'est le nom du paramètre ou de l'attribut

```
public MaClasse
 {
    public MaClasse(int a, int b) {...}

    public MaClasse (int c)
     {
        this(c,0);
     }

    public MaClasse ()
     {
        this(10);
     }
}
```

Appelle le constructeur MaClasse(int a, int b)

Appelle le constructeur MaClasse(int c)

Référence à la superclasse

- Le mot réservé **super** permet de faire référence au constructeur de la superclasse directe mais aussi à d'autres informations provenant de cette superclasse.

```
class Employe extends Personne
{
 private float salaire;
 public float calculePrime()
 {
  return (salaire * 0,05);
 }
 // ...
}
```

Appel à la méthode calculPrime() de la superclasse de Cadre

```
class Cadre extends Employe
{
 public float calculePrime()
 {
  return (super.calculePrime() / 2);
 }
 // ...
}
```

Opérateur instanceof

- L'opérateur **instanceof** permet de savoir si une instance est instance d'une classe donnée. Il renvoie une valeur booléenne.

```
Personne jean = new Employe();


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