Chap. 4 : héritage

Héritage

Licence professionnelle

UV G5a

Pascal Divoux ; Cedric Wemmert

Chap. 4 : héritage

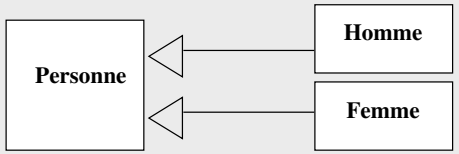
➢ Spécialisation

➔ Relation de spécialisation « est-un »

➔ Hiérarchie de classification stable qui reflète la connaissance humaine *(cf classification de Linné)*

➔ L'héritage est un concept de classe, pas un concept d'objet

➔ Notation UML :



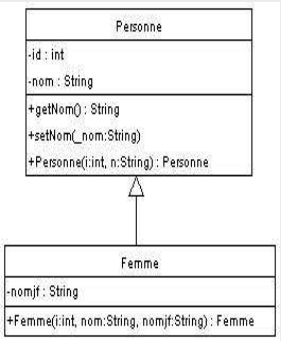
➔ Autres exemples :

- ***Une fourmi est un insecte***
- ***Un commercial est un employé***

Chap. 4 : héritage

❁ En java : l 'extension

➔ La définition d'une sous-classe se fait par " extension " (**extends**) d'une classe existante. Elle hérite des attributs et des méthodes (non privés) de sa super-classe. La structure de la sous-classe est donc composée de ses propres attributs et de ceux provenant de la super-classe, de même pour les méthodes.




```
public class Personne{
        int id;
        String nom;
        public Personne (int i,String n){
                this.id=i;
                this.nom=n;
        }
}
public class Femme extends Personne {
        String nomjf;
        ...
}
```


Chap. 4 : héritage

➔L'opérateur `instanceof` permet de déterminer la classe d'une instance (résultat booléen).

```
Homme h= new Homme();
if (h instanceof Homme) ... // true
if (h instanceof Personne) ... // true
```

➔ this désigne l 'instance de classe courante

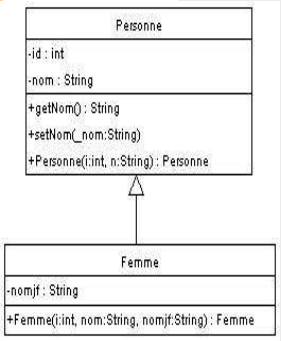
➔ super référence la super-classe de la classe courante

➔ Une classe ne peut hériter directement que d 'une autre classe (pas d 'héritage multiple)

➔Les classes `final` ne peuvent être dérivées.

➔Toutes les classes héritent directement ou non de la classe « Object »

Constructeurs d 'une sous-classe



➔ toute instance d 'une sous-classe est aussi une instance de sa super-classe

➔ donc la construction d 'une sous-classe doit commencer par construire la super-classe

➔ Le désignateur **super(...)** permet d'invoquer dans le constructeur de la sous-classe, le constructeur de sa super-classe. Java impose que tout constructeur appelle explicitement ou implicitement un constructeur de sa super-classe.

```
public class Femme extends Personne {
  String nomjf;
  public Femme (int i,String n,
  String njf){
        super(i,n);
        this.nomjf=njf;
  }
}
```


➔Si la première instruction d'un constructeur n'est pas `super()` ou `this()`, un appel à `super()` est automatiquement ajouté avant la première instruction. Dans ce cas, il faut qu'un constructeur sans paramètre existe dans la super-classe.

➔Un objet de type x peut etre manipulé par une référence de n 'importe quel super-type de x

```
Personne p1 = new Homme(123,``Dupont'');
Personne p2 = new Femme (124,``Dupont'',``Dulac'');
```

mais elle ne peut utiliser que les propriétés du type déclaré :

```
p2.nom //OK
p2.nomjf //pas OK
```

pour accéder aux propriétés de l 'objet réel il faut faire une conversion explicite (un « cast ») sous la responsabilité du programmeur

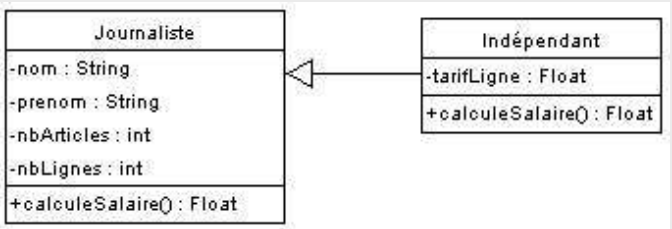
```
Femme f=(Femme)p2;
f.nomjf  //OK
Femme f=(Femme)p1 //OK à la compil !!!
```


Redéfinition et surcharge

➔ Une sous-classe peut redéfinir une méthode héritée : il faut la réécrire en conservant la même signature (mêmes paramètres) et le même type de retour.

- *La méthode « calculeSalaire » de « Indépendant » redéfinit la méthode de sa super-classe « Journaliste » car l 'indépendant n 'a pas le même mode de calcul de paie (tarifLigne)*
- *La méthode invoquée à l 'exécution est celle du type réel de l 'instance*

```
Journaliste j=new(Independant);
Float sal=j.calculeSalaire();//
```



Redéfinition et surcharge

➔Une méthode peut être surchargée (dans la même classe ou une classe fille) si on garde le même nom et le même type de retour mais pas les mêmes paramètres (pas le même nombre ou pas le même type).

- *La méthode invoquée à l 'exécution est celle qui correspond au nombre et aux types des paramètres de l 'appel*
- *Les constructeurs sont fréquemment surchargés*

```
JOptionPane();
JOptionPane(Object message);
JOptionPane(Object message,int typeMess, int
  typeOption, Icon icone);
...
```