L'HÉRITAGE

1. Définition

L'héritage permet de créer une nouvelle classe (appelée classe fille) à partir d'une classe existante (appelée classe mère).

La classe fille hérite des attributs et des méthodes de sa classe mère.

Synonyme de classe fille : sous-classe.
Synonyme de classe mère : super-classe.

2. Mise en œuvre

Pour préciser qu'une classe hérite d'une autre classe on utilise le mot clé *extends*.

Voir *demo_heritage.exemple1*.

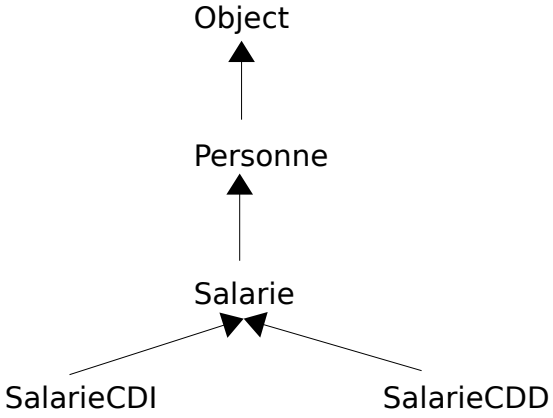
3. Conséquence

Une classe fille n'a accès qu'à ce qui est déclaré publique dans la classe mère. Elle possède néanmoins les attributs privés de la classe mère mais elle ne peut pas les manipuler directement.

4. Hiérarchie de classes

En Java, seul l'héritage simple existe c'est-à-dire qu'une classe ne peut hériter que d'une seule classe.
Lorsque l'on ne précise pas qu'une classe hérite d'une autre, elle hérite automatiquement de la classe *Object*.

En Java, il existe donc une hiérarchie de classes dont la racine est la classe *Object* car toutes les classes sont liées par une relation d'héritage.



La flèche se lit « hérite de ». On peut aussi la lire « est un ».

On peut dire qu'un *Salarié* est une *Personne*, qu'une *Personne* est un *Object*.

Remarque : en Java, l'héritage multiple (capacité pour un classe d'hériter de plusieurs classes) n'existe pas contrairement à d'autres langages comme le C++.

Néanmoins, il existe un mécanisme très proche de l'héritage multiple : les interfaces. Nous examinerons ce mécanisme ultérieurement.

5. Constructeur

Le constructeur d'une classe fille doit toujours invoquer le constructeur de sa super classe comme première instruction. Si le code source n'inclut pas d'appel de ce type, le compilateur tente d'insérer un appel automatiquement.

Voir *demo_heritage.exemple2*.

6. Type d'une variable et type d'une donnée

Quand une variable est définie, un type lui est donné. Par exemple : `Personne p`. Cela signifie que la variable *p* ne peut référencer que des données de type *Personne* ou des données dont le type est un sous-type de *Personne*.

Exemple :

```
Personne p = new Personne() ;
p = new Salarie() ; // correct
p = new SalarieCDI() ; // correct
p = new SalarieCDD() ; // correct
p = new Object() ; // provoque une erreur à la compilation
```

7. Redéfinition (*overriding*)

Une classe fille peut redéfinir une implémentation de méthode. Pour cela la classe fille déclare une méthode avec la même signature que dans la classe mère mais avec un corps différent. La méthode remplaçante a priorité pour les appels de méthode sur les objets de cette sous-classe.

On pourra préciser (mais ce n'est pas une obligation) que l'on redéfinit une méthode en utilisant l'annotation *@Override* devant la méthode redéfinie.

Remarque : il ne faut pas confondre la redéfinition (*overriding*) avec la surcharge (*overloading*).

Voir *demo_heritage.exemple3*.

Exemple de la méthode *toString()* de la classe *Object* et utilisation de la méthode *println*.

8. Polymorphisme

Voir *demo_polymorphisme*.

Le type de *p* au moment de la déclaration est *Personne*. Ce type est appelé le type statique de *p*. Il est donné une fois pour toute et il ne peut pas changer.

p peut référencer des données de type *Personne* bien sûr, mais aussi tous les sous-types (directs ou indirects) de *Personne*. *p* peut donc référencer des données de type *Salarie* ou *Apprenti*.

Le type de la donnée référencée par *p* peut ainsi varié au cours du temps. Le type de la donnée référencée par *p* est appelé le *type dynamiqu*e de *p*.

La méthode *afficher* existe sous trois formes différentes (elle a été redéfinie). Elle est polymorphe.

On dit aussi que l'appel de la méthode *afficher* est polymorphe. Au moment de l'appel de cette méthode, le machine virtuelle (la JVM) déterminera la bonne méthode *afficher* (celle de *Personne*, celle de *Salarie* ou celle de *Apprenti*) à utiliser en fonction du type de la donnée référencée par *p* c'est-à-dire son type dynamique.