

L'héritage et le polymorphisme en Java

Pour signifier qu'une classe fille hérite d'une classe mère, on utilise le mot clé `extends`

```
class fille extends mère
```

En java, toutes les classes sont dérivées de la classe **Object**.

La classe Object est la classe de base de toutes les autres. C'est la seule classe de Java qui ne possède pas de classe mère. Tous les objets en Java, quelle que soit leur classe, sont du type Object. Cela implique que tous les objets possèdent déjà à leur naissance un certain nombre d'attributs et de méthodes dérivées d'Object.

Dans la déclaration d'une classe, si la clause `extends` n'est pas présente, la surclasse immédiatement supérieure est donc Object.

Les membres protégés

Les sous-classes n'ont pas accès aux membres private de leur classe mère.

Si un attribut de la classe mère est déclaré `private`, cet attribut n'est pas accessible par les sous-classes. Seule la classe qui possède un membre `private` peut y accéder.

Remarque: les membres privés de la classe mère sont bien hérités par les classes filles. Chaque instance des classes filles possède les membres privés de leur classe mère, mais ceux-ci ne sont accessibles qu'à travers les accesseurs et/ou les méthodes de la classe mère qui les utilise.

ex:

```
public class Ville
```

```
{
    private String nom;
    private int nbHab;
    ...
}
```

```
public class Capitale extends Ville
```

```
{
    private String pays;

    public afficheCapitale()
    {
        System.out.println(nom + " est la Capitale de "+ pays);
    }
    ...
}
```

Erreur!
nom est un attribut
private de Ville et n'est
donc pas accessible dans
la classe Capitale

Si on veut qu'un attribut ou une méthode soit encapsulé pour l'extérieur mais qu'il soit accessible par les sous-classes, il faut le déclarer **protected** à la place de `private`.

```
public class Ville
```

```
{
    protected String nom;
    protected int nbHab;
    ...
}
```

```
public class Capitale extends Ville
```

```
{
    private String pays;

    public afficheCapitale()
    {
        System.out.println(nom + " est la Capitale de "+ pays);
    }
    ...
}
```

OK
nom est accessible par la sous-classe car il
est protected dans la classe mère

Redéfinition des méthodes héritées

Une méthode de la classe mère peut être implémentée différemment dans une classe fille: la méthode est dite redéfinie. Aucun signe n'indique en java qu'une méthode est redéfinie (contrairement à Pascal ou à C++).

La redéfinition d'une méthode dans une classe fille cache la méthode d'origine de la classe mère. Pour utiliser la méthode redéfinie de la classe et non celle qui a été implémentée dans la classe fille, on utilise le mot-clé **super** suivi du nom de la méthode.

ex:

`super.machin()` fait appel à la méthode `machin()` implémentée dans la classe mère et non à l'implémentation de la classe fille.

Utilisation d'un constructeur de la classe mère

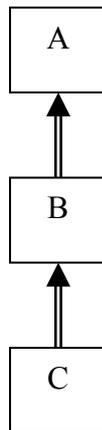
On peut faire appel au constructeur de la classe mère en utilisant le mot-clé **super()** avec les paramètres requis. `super` remplace le nom du constructeur de la classe mère.

L'instruction `super()` doit être obligatoirement la première instruction des constructeurs de la classe fille.

Tout constructeur d'une classe fille appelle forcément l'un des constructeurs de la classe mère : si cet appel n'est pas explicite, l'appel du constructeur par défaut (sans paramètre) est effectué implicitement.

Ainsi, un objet dérivé est toujours construit sur la base d'un objet de sa classe mère.

Exemple:



Pour construire un objet de la classe C, il faut d'abord construire un objet de la classe B. Mais pour créer un objet de la classe B, il faut d'abord créer un objet de la classe A. Et pour créer un objet de la classe A, il faut d'abord créer un objet de la classe Object.

Les constructions s'effectuent donc dans cet ordre:

construction d'un Object

construction d'une instance de A

construction d'une instance de B

construction d'une instance de C

(cela ne vous fait-il pas penser au cours sur les piles???)

Le mot clé final

Un attribut déclaré **final** est un attribut **constant**. Il faut l'initialiser dès sa déclaration (comme les `const` en C++). Par convention, un attribut final est écrit en majuscule.

ex: `final int MAX = 1000;`

Une classe final est une classe **qui ne peut pas avoir de filles**. Une classe final ne peut pas être étendue: le mécanisme d'héritage est bloqué. Mais une classe final peut évidemment être la fille d'une autre classe (non final!).

Une méthode final est une méthode qui ne peut pas être redéfinie dans les sous-classes.

Les classes abstraites

Une classe abstraite est totalement l'opposé d'une classe final:

Une classe abstraite est une classe qui ne peut pas être instanciée directement. Elle n'est utilisable qu'à travers sa descendance. Une classe abstraite doit toujours être dérivée pour pouvoir générer des objets.

Une classe abstraite est précédée du mot clé `abstract`

`abstract class classeabstraite`

L'utilité d'une classe abstraite est de permettre le regroupement (la factorisation) des attributs et méthodes communs à un groupe de classes.

Dans une classe abstraite, on peut trouver des **méthodes abstraites**, c'est à dire des méthodes **qui n'ont pas d'implémentation possible dans la classe abstraite, mais qui doivent obligatoirement être implémentées différemment dans toutes les classes filles**. L'intérêt des méthodes abstraites vient du fait que l'on peut les appeler de la même façon pour tous les objets dérivés : on est en plein dans le polymorphisme. (vous comprendrez cet intérêt quand vous utiliserez des méthodes abstraites en TP).

Une classe qui possède une (ou plusieurs) méthode abstraite est obligatoirement abstraite.

Un exemple complet : Héritage Ville → Capitale

Voici une classe Ville, qui sera étendue par la suite par une classe Capitale

```
class Ville
{
    private String nom;    //le nom ne sera accessible que par la classe Ville, et pas par la classe Capitale
    protected int nbHab; //le nombre d'habitant sera accessible par la classe Capitale

    public Ville(String leNom)
    {
        nom = leNom.toUpperCase();    //ainsi tous le noms de ville seront en majuscule
        nbHab = -1;    //-1 signifie que le nombre d'habitant est inconnu
    }
    public Ville (String leNom, int leNbHab)
    {
        nom = leNom.toUpperCase();
        if (leNbHab < 0)
        {
            System.out.println("Un nombre d'habitant doit être positif.");
            nbHab = -1;
        }
        else
            nbHab = leNbHab;
    }

    public String getNom()
    {
        return nom;
    }
    //pas d'accessor en écriture pour le nom → il est impossible de changer le nom d'une ville

    public int getNbHab()
    {
        return nbHab;
    }
    public void setNbHab(int nvNbHab)
    {
        if (nvNbHab < 0)
            System.out.println("Un nombre d'habitant doit être positif. La modification n'a pas été prise en compte");
        else
            nbHab = nvNbHab;
    }

    public String presenteToi()
    {
        String presente = "Ville "+ nom +" nombre d'habitants ";
        if (nbHab == -1)
            presente = presente + "inconnu";
        else
            presente = presente + " " + nbHab;
        return presente;
    }
}
```

```
class Capitale extends Ville
{
    private String pays;
    //constructeurs
    public Capitale(String leNom, String lePays)
    {
        super(leNom); //appel du constructeur de Ville. nbHab est initialisé à -1 par ce constructeur
        pays = lePays;
    }
    public Capitale(String leNom, String lePays, int leNbHab)
    {
        super(leNom, leNbHab);
        pays = lePays;
    }

    //accesseurs supplémentaires
    public String getPays()
    {
        return pays;
    }
    public void setPays(String nomPays)
    {
        pays = nomPays;
    }

    //méthode presenteToi() redéfinie
    public String presenteToi()
    {
        String presente = super.presenteToi();
        presente = presente + " Capitale de "+ pays;
        return presente;
    }
}

//Classe de test
public class testVille
{
    public static void main(String args[])
    {
        Ville v1 = new Ville("Lyon", 1500000);
        Ville v2 = new Ville("Bobigny");
        Capitale c1 = new Capitale("Paris", "France", 10000000);
        Capitale c2 = new Capitale("Ouagadougou", "Burkina-Faso");
        System.out.println(v1.presenteToi());
        System.out.println(v2.presenteToi());
        System.out.println(c1.presenteToi());
        System.out.println(c2.presenteToi());
    }
}
```

Exécution:

```
Ville Lyon nombre d'habitants = 1500000
Ville Bobigny nombre d'habitants inconnu
Ville Paris nombre d'habitants 10000000 Capitale de France
Ville Ouagadougou nombre d'habitants inconnu Capitale de Burkina-Faso
```

Le transtypage et le polymorphisme

Le transtypage (conversion de type ou cast en anglais) consiste à modifier le type d'une variable ou d'une expression.

Par exemple, il est possible transtyper un int en double.

❑ transtypage des types primitifs

Il existe deux types de transtypages: le **transtypage implicite** et le **transtypage explicite**.

➤ **Le transtypage implicite est traité automatiquement par le compilateur** lors d'une affectation ou du passage d'un paramètre effectif. Un transtypage peut être implicite **si le type cible a un plus grand domaine que le type d'origine (gain de précision)**

Par exemple, il est possible d'affecter directement un int à un double, ou un byte en short, sans expliciter de transtypage : le compilateur effectue automatiquement la conversion.

➤ En revanche lorsqu'on veut **convertir une variable ou une expression dans un type qui lui fait perdre de la précision (domaine de valeurs plus restreint)**, il faut réaliser un **transtypage explicite** (sinon erreur du compilateur).

Par exemple, convertir un float en int fait perdre les chiffres après la virgule, convertir un long en short donne des résultats aberrants pour les grands entiers (qui ne peuvent pas être représentés avec un short) : dans ces cas, il faut donc réaliser un cast explicite. De ce côté, Java est un langage bien plus restrictif que C++, où même les transtypages avec perte de précision, peuvent être implicites.

Pour réaliser un cast explicite, on met entre parenthèses le nom du type dans lequel on veut convertir suivi du nom de la variable (ou de l'expression entre parenthèses) qu'on veut transtyper.

ex :

```
double d; int i;
```

```
i = (int) d; //transtypage d'un double en int pour affectation
```

❑ Le transtypage de références d'objets

Il est possible de convertir un objet d'une classe en un objet d'une autre classe si les classes ont un lien d'héritage (encore une fois, on utilise le mot objet par abus de langage : ce sont les références aux objets et non les objets eux-mêmes qui sont transtypés).

Le transtypage d'un objet dans le sens fille → mère est implicite.

En revanche, le transtypage dans le sens mère → fille doit être explicite et n'est pas toujours possible.

➤ **Transtypage implicite (sens fille → mère)**

Il est toujours possible d'utiliser une référence de la classe mère pour désigner un objet d'une classe dérivée (fille, petite-fille et toute la descendance). Il y a alors transtypage implicite de la classe fille vers la classe mère.

Cela est logique si on se dit qu'un objet dérivé EST un objet de base.

Exemple : une Capitale EST une Ville, donc on peut utiliser une référence de type Ville pour désigner une Capitale → il y a transtypage implicite d'une Capitale en Ville.

Par contre, une Ville n'est pas forcément une Capitale : donc on ne peut pas transtyper une Ville en Capitale, autrement utiliser une référence de Capitale pour désigner une Ville.

☞ Remarquez que contrairement au transtypage des types primitifs, le transtypage implicite de références entraîne une perte de précision (le type Ville étant moins précis que le type Capitale)

Exemple d'utilisation du transtypage implicite :

```
Ville v;
```

```
Capitale c = new Capitale("Paris", "France");
```

v = c; //transtypage implicite d'une Ville en Capitale

L'instruction d'affectation `v = c` implique un transtypage implicite. La référence `v` du type `Ville` est alors utilisée pour désigner un objet de type `Capitale`. L'objet désigné par `c`, qui est du type `Capitale` est donc transtypé en `Ville`.

Donc on ne pourra pas utiliser `v` pour appeler une méthode spécifique de la classe `Capitale` (car `v` est une référence sur une `Ville`, qui ne peut pas appeler des méthodes de la classe `Capitale`). Il faudrait pour cela effectuer un transtypage explicite de `v` en `Capitale`.

➤ **Transtypage explicite (sens mère→fille)**

Le transtypage explicite des références est utilisé pour convertir le type d'une référence dans un type dérivé. C'est logique si l'on se dit qu'un objet d'une classe mère n'est pas un objet de ses classes filles (une `Ville` n'est pas une `Capitale`).

Par exemple, si l'on voulait utiliser la référence `v` (qui est du type `Ville` mais qui désigne une `Capitale`) pour invoquer une méthode spécifique de la classe `Capitale`, il faudrait réaliser un transtypage de `v` en `Capitale` (sinon, le compilateur refuserait : le transtypage dans le sens mère→fille n'étant pas implicite).

```
String lePays;
lePays = (Capitale)v.getPays();
```

Si on avait écrit `lePays = v.getPays()`, le compilateur aurait généré une erreur car `getPays()` n'est pas une méthode de la classe `Ville` (donc faire partie `v`).

☞ Un transtypage explicite n'est pas toujours possible, même si les classes ont un lien de parenté. C'est au programmeur de vérifier que son transtypage n'engendrera pas d'erreur, c'est-à-dire que l'objet est bien effectivement du type dans lequel on transtype la référence.

`v` peut être transtypé en `Capitale` car `v` désigne effectivement un objet de type `Capitale`.

Exemple de transtypage explicite impossible : créer un objet `Ville` à partir d'une référence de type `Capitale` car dans ce cas, `c` ne peut pas désigner une capitale (il n'y a pas de pays)

```
Capitale c = new Ville("Bruxelles");
```

❑ **Le polymorphisme comme choix dynamique du code à l'exécution**

Et si on voulait invoquer la méthode `presenteToi()` avec la référence `v` ?

```
v.presenteToi();*
```

Quelle implémentation de `presenteToi()` s'exécuterait alors ?

Celle de `Ville` ... qui afficherait :

```
Ville Paris nombre d'habitants inconnu
```

ou celle de `Capitale` ? qui afficherait

```
Ville Paris nombre d'habitants inconnu Capitale de France
```

Réponse : c'est l'implémentation de `Capitale` qui est exécutée, même si `v` est de type `Ville`

Pourquoi ?

A l'invocation d'une méthode, le choix de l'implémentation à exécuter ne se fait pas en fonction du type déclaré de la référence à l'objet, mais en fonction du type réel de l'objet. `v` désigne un objet de type `Capitale`, même si c'est une référence de type `Ville`: donc c'est la méthode implémentée dans `Capitale` qui est exécutée.

Ce mécanisme montre un aspect important du polymorphisme :

le choix du code à exécuter (pour une méthode polymorphe) ne se fait pas statiquement à la compilation mais dynamiquement à l'exécution.

* Important : On supposera ici pour alléger l'écriture, que la méthode `presenteToi` est une procédure qui permet un affichage et non une fonction qui renvoie une chaîne de caractère comme dans l'exemple précédent.

□ Exemple d'utilisation du polymorphisme et du transtypage avec un Vector

Dans un objet de type Vector (qui permet de mémoriser une liste d'objets), les éléments sont des références de type Object. Mais grâce au mécanisme de transtypage implicite, un Vector peut contenir des références à des objets de n'importe quelle classe. En effet, la classe Object étant la classe mère de toutes les autres, (directement ou indirectement), tout élément de Vector (référence de type Object) peut être transtypé en n'importe quelle classe (forcément dérivé d'Object)

Exemple :

```
Vector maListe = new Vector( );
Ville v1 = new Ville("Lyon", 2000000);
Ville v2 = new Ville("Dijon");
Capitale c1 = new Capitale("Alger", "Algerie");
maListe.addElement(v1);           //transtypage implicite de Object (type du paramètre) en Ville
maListe.addElement(v2);
maListe.addElement(c1);
```

Mais ensuite, pour utiliser les éléments d'un Vector (pour pouvoir invoquer leurs méthodes propres notamment), il faut indiquer explicitement la classe de ces éléments par un transtypage explicite.

~~maListe.elementAt(0).presenteToi();~~
 provoque une erreur car la méthode ElementAt() de Vector retourne un objet du type Object et non du type Ville.

Pour pouvoir invoquer la méthode presenteToi(), il faut transtyper l'élément extrait du vecteur par la méthode elementAt() en Ville :

```
((Ville)(maListe.elementAt(0))).presenteToi( );
```

Object
Ville

Comme une Capitale est une Ville, il est possible d'invoquer sa méthode presenteToi() même à travers une référence de type Ville et non de type Capitale. Ainsi, on va pouvoir utiliser une boucle sur tout le vecteur, transtyper tous les éléments en Ville et leur envoyer le message presenteToi(). Les Capitales se présenteront comme telles et non comme des villes simples.

```
for (int i = 0; i < 3; i++)
{
    ((Ville)(maListe.elementAt(i))).presenteToi( );
}
```

C'est la méthode presenteToi() de Capitale qui est exécutée pour le troisième élément, car le choix du code d'une méthode se fait selon le type réel de l'objet et non selon le type de la référence qui le désigne.