# Chapitre 8

# Héritage

## 1 Le concept d'héritage en POO

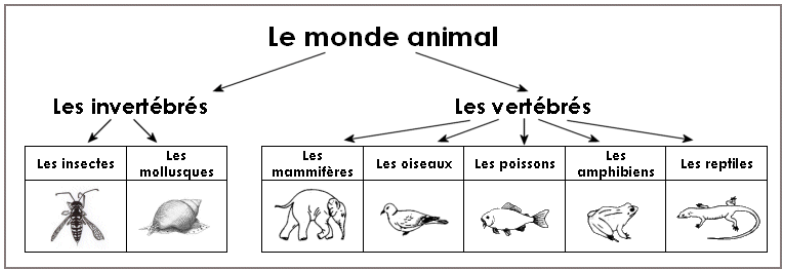
L'héritage est l'un des concepts fondamentaux de la POO. Son principe est de définir un nouvel objet en réutilisant une définition plus générale d'un objet proche, et en la spécialisant au cas particulier du nouvel objet. L'héritage permet de faire l'économie de tout redéfinir en ne définissant que ce qui est spécifique du nouvel objet. Ce mécanisme permet aussi au nouvel objet de bénéficier « gratuitement » de tout ce qui a été défini par ses ancêtres.

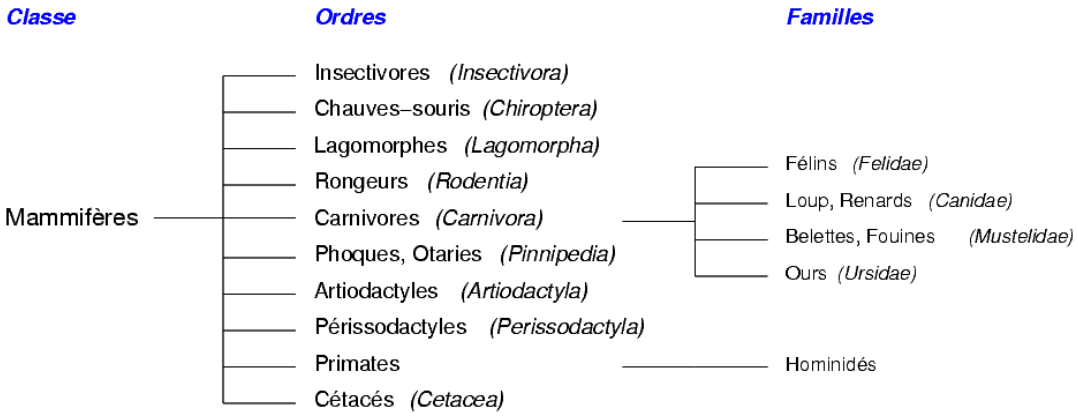
Considérons ainsi deux types d'objets $A$ et $B$, tels que que les objets de type $B$ soient comme les objets de type $A$, à quelques détails près ... Plutôt que de redéfinir complètement $B$ indépendamment de $A$, on peut être tenté de ne préciser que ce qui, dans $B$, est différent de $A$. Dans ce cas, on dit que $B$ *hérite* de $A$.

### 1.1 Sous-classement

Plus précisément, le mécanisme mis en jeu par l'héritage est celui de *sous-classement*, qui est à la base de nombreux systèmes de classification. Une sous-classe possède toutes les caractéristiques de la classe dont elle hérite, mais possède en plus des caractéristiques qui lui sont propres. Ce mécanisme permet des descriptions qui vont du général au particulier.

**Exemple.** La classification (simplifiée) du règne animal.






**Remarque.** On peut trouver aussi des exemples de sous-classement dans le domaine informatique : un fichier texte est un cas particulier de fichier. Une clé USB est un cas particulier de dispositif de stockage externe USB, qui est lui-même un cas particulier de dispositif de stockage externe, ...

## 1.2 Relation d'héritage

Ce qu'il faut retenir de ce genre de schéma, c'est que :

**La relation d'héritage est une relation « est un ».**

Ainsi, un homme *est un* hominidé ; un hominidé *est un* primate ; un primate *est un* mammifère ; etc.

Cette relation est transitive : un homme *est un* primate ; un homme *est un* mammifère ; etc.

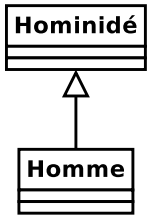
Cette relation n'est pas symétrique : tous les primates ne sont pas des hommes.

# 2 Exemple, syntaxe et sémantique de l'héritage

**Vocabulaire et notation.** Voyons Homme, Hominidé, etc. comme des classes. On dit que la classe Homme *hérite de* la classe Hominidé. Dans cette relation, on dit que Homme est une *sous-classe* de Hominidé. Inversement, on dit que Hominidé est la *super-classe* de Homme. Une sous-classe est également appelée une classe *dérivée*.

En UML, la relation d'héritage est indiquée par une flèche dirigée de la sous-classe vers la super-classe.

**Exemple.**




En PDL++, on utilisera les mots clés hérite de. Ainsi, pour indiquer qu'une classe $B$ hérite d'une classe $A$, on écrira en PDL++ :

```
classe Humain hérite de Hominide {
  . . .
}
```

En java, on utilise le mot clé extends :

```
class Humain extends Hominide {
  . . .
}
```

## 2.1 Exemple : héritage de la classe `Horaire`

On rappelle en figure 8.1 le code de la classe `Horaire`, tel que donné au chapitre 7.

```
classe Horaire {
      entier h, m, s;

      rien setHoraire(entier h, entier m, entier s)
      Si (h >= 0 ET h < 23 ET m >= 0 ET m < 60 ET s >= 0 ET s < 60) Alors
            moi.h ← h;
            moi.m ← m;
            moi.s ← s;
      fin
}
```

FIG. 8.1 – Rappel : le code de la classe `Horaire` en PDL++

**Remarque.** Pour des raisons pédagogiques dans ce chapitre, on fait ici abstraction du caractère en principe privé des attributs. On revient donc à une version où les attributs $h$, $m$ et $s$ ne sont pas privés.

On souhaite maintenant définir un horaire plus précis, en ce sens qu'en plus des heures, minutes et secondes, on intègre également les millisecondes. Un horaire précis *est un* horaire : cela indique que l'horaire précis peut être défini par héritage de horaire. Attention, un horaire *n'est pas* un horaire précis : il n'en possède pas les millisecondes.

Ainsi, on peut définir la classe HorairePrecis par héritage de la classe Horaire. C'est ce qui est réalisé dans la figure 8.2.

```
classe HorairePrecis hérite de Horaire {
        entier ms ;

        rien setHoraire(entier h, entier m, entier s)
        Si (h >= 0 ET h < 23 ET m >= 0 ET m < 60 ET s >= 0 ET s < 60) Alors
                moi.h ← h ;
                moi.m ← m ;
                moi.s ← s ;
        FSi
        moi.ms ← 0 ;
        fin

        rien setHoraire(entier h, entier m, entier s, entier ms)
        Si (h >= 0 ET h < 23 ET m >= 0 ET m < 60 ET s >= 0 ET s < 60) Alors
                moi.h ← h ;
                moi.m ← m ;
                moi.s ← s ;
        FSi
        Si (ms >= 0 ET ms < 1000) Alors
                moi.ms ← ms ;
        FSi
        fin
}
```

FIG. 8.2 – La classe `HorairePrecis`, définie par héritage de la classe `Horaire`

**Remarque.** On a fait le choix que si on ne précise pas le nombre de millisecondes lors de l'appel à setHoraire(...), celui-ci prend par défaut la valeur 0.

## 2.2 Héritage des attributs

On remarque que la classe `HorairePrecis` ne déclare pas d'attributs $h$, $m$ et $s$ pour les heures, minutes et secondes. En effet, une conséquence du fait que `HorairePrecis` hérite de `Horaire` est qu'elle en possède automatiquement tous les attributs (non privés). Les attributs de `Horaire` ne doivent donc pas être redéclarés dans HorairePrecis. Ils peuvent être manipulés par les méthodes de HorairePrecis car ce sont bien des attributs (ils sont hérités) de la classe HorairePrecis.

HorairePrecis déclare seulement le nouvel attribut $ms$ qui enregistre les millisecondes.

## 2.3 Héritage des méthodes

De même que pour les attributs, une classe dérivée hérite également des méthodes non-privées de sa super-classe. Ici, HorairePrecis hérite de la méthode setHoraire(entier h, entier m, entier s) de Horaire. Pourtant, HorairePrecis déclare elle même une méthode setHoraire(entier h, entier m, entier s) ! Est-ce une erreur ?

La réponse est non. La méthode setHoraire(entier h, entier m, entier s) de Horaire existe toujours dans HorairePrecis (elle est héritée). Mais elle est *masquée* par la méthode setHoraire(entier h, entier m, entier s) donnée dans HorairePrecis. Il s'agit d'une *redéfinition* de méthode.

## 2.4 Redéfinition des méthodes

La méthode setHoraire(entier h, entier m, entier s) de HorairePrecis a le même nom et la même liste de paramètres que celle de Horaire car elle poursuit le même objectif, mais pour un horaire précis. Elle est *redéfinie*.

**Définition 8.1.** *Redéfinir une méthode consiste à donner dans une sous-classe une nouvelle définition d'une méthode adaptée à la sous-classe, en utilisant la même signature pour cette méthode que dans la super-classe.*

La redéfinition ne doit pas être confondue avec la surcharge, qui consiste à donner une nouvelle définition d'une méthode mais pour une signature différente.

Dans notre exemple, on a à la fois de la redéfinition *et* de la surcharge : setHoraire(entier h, entier m, entier s) redéfinit la même méthode de la super-classe, et setHoraire(entier h, entier m, entier s, entier ms) surcharge la méthode setHoraire(entier h, entier m, entier s) de HorairePrecis, pour préciser les millisecondes si nécessaire, et ne pas les mettre automatiquement à 0.

**Remarque.** La méthode setHoraire entier h, entier m, entier s, entier ms) pourrait s'écrire en invoquant la méthode redéfinie :

```
rien setHoraire(entier h, entier m, entier s, entier ms)
  setHoraire(h, m, s) ;
  Si (ms >= 0 ET ms < 1000) Alors
        moi.ms ← ms ;
  FSi
  fin
```

## 2.5 Le mot clé **super**

### 2.5.1 `super` pour désigner la super-classe

Considérons une sous-classe (par exemple HorairePrecis) qui redéfinit une méthode de sa super-classe (par exemple Horaire). La méthode héritée est masquée par cette nouvelle définition. Il est pourtant toujours possible de l'invoquer, en préfixant cet appel par le mot-clé super. Ce mot-clé (utilisé en PDL++ et en Java) contient une *référence* à : « la super classe de moi ».

Ainsi, la redéfinition de setHoraire(entier h, entier m, entier s) aurait pu s'écrire par :

```
rien setHoraire(entier h, entier m, entier s)
        super.setHoraire(h,m,s) ;
        moi.ms = 0 ;
  fin
```

Sans ce mot clé, l'invocation de setHoraire(...) depuis la redéfinition de setHoraire(...) serait un appel récursif !

**Remarque.** Bien que ce ne soit pas du tout recommandé, il est possible à une classe dérivée de *masquer* les attributs de sa super-classe. Il suffit pour cela de donner à un attribut de la classe dérivée le même nom qu'à l'un des attributs de la super-classe. Le mot-clé `super` permet alors de désigner l'attribut de la super classe plutôt que celui de la classe dérivée.

### 2.5.2 Le constructeur de la super classe

Utilisé comme un appel de méthode, super(...) permet de désigner le constructeur de la super classe. Ce point est précisé dans la partie 4 de ce chapitre.

## 2.6 Classes déclarées **final**

Si on ne veut pas qu'une classe puisse être héritée, il suffit de la déclarer final.

**Exemple.** public final class A { ... }

## 2.7 Effet des modificateurs de visibilité

Les attributs et méthodes privés d'une classe ne sont pas accessibles depuis une sous-classe de cette classe.

Il existe un modificateur nommé protected. Il rend les attributs et méthodes accessibles à toutes les classes dérivées.

# 3 Héritage en série

## 3.1 Pas d'héritage multiple en Java

Nous avons vu qu'il était possible à une classe $B$ d'hériter d'une classe $A$. Rien n'interdit bien sûr à une classe $C$ d'hériter elle-même de la classe $B$. Autrement dit, une sous-classe peut elle même être sous-classée.

Une classe donnée peut être sous-classée plusieurs fois. Par contre, il n'est pas possible à une classe d'avoir plusieurs super-classes en Java. Le fait de pouvoir hériter directement de plusieurs classes est connu sous le nom d'héritage multiple en POO. Un langage comme C++ autorise l'héritage multiple, mais le langage Java ne le permet pas (dans le but de rendre le langage plus clair).

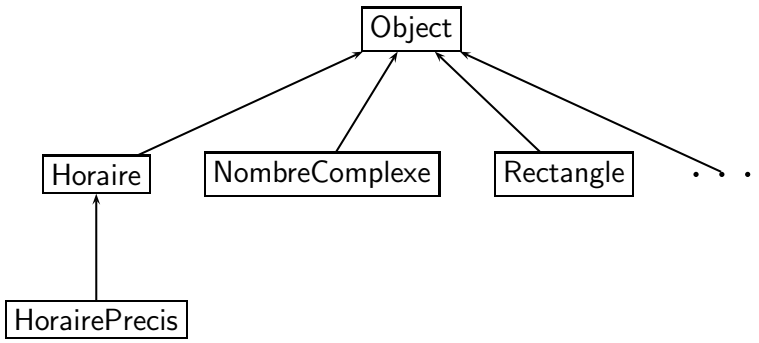
## 3.2 Hiérarchie des classes

Il est à noter qu'en Java, toute classe hérite nécessairement d'une super-classe, même lorsque le programmeur ne le déclare pas explicitement au moyen du mot clé `extends`. En effet, Java possède une classe de base intitulée `Object`. Toute classe non rattachée explicitement à une super-classe (grâce au mot-clé `extends`) hérite automatiquement de la classe `Object`. Par transitivité, il en résulte que toute classe Java hérite, directement ou indirectement, de la classe `Object`.

**Remarque.** C'est grâce à ce mécanisme que toute classe Java se retrouve automatiquement dotée d'une méthode `toString()`. En effet, `toString()` est définie dans la classe `Object`, et donc toute classe Java en hérite, et peut la redéfinir. La classe `Object` définit également d'autres méthodes que nous ne détaillons pas ici, mais que vous pouvez consulter dans l'API Java.

Ainsi, le mécanisme d'héritage présenté dans ce chapitre permet de placer toute classe définie par héritage au sein d'une arborescence représentant la hiérarchie des classes. La racine de cette arborescence est toujours la classe `Object`.

**Exemple.**



# 4 Héritage et chaînage des constructeurs

Le mécanisme d'héritage est assuré *physiquement* par le fait que toute instance d'une classe dérivée contient en elle une instance de sa super-classe. De ce fait, l'instance de la classe dérivée connaît non seulement les attributs et les méthodes qui lui sont propres, mais également ceux qui sont hérités.

**Exemple.** Soient les deux classes suivantes

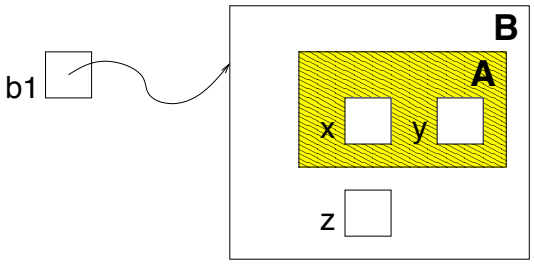
```
classe A {
      entier x, y
}
```

```
classe B hérite de A {
      entier z
}
```

et le programme suivant :

```
programme principal
      b1 ← créer B()
fin
```

L'instanciation de $b1$ (par b1 ← créer B()) créée la situation suivante en mémoire :




Lorsqu'une classe $B$ hérite d'une classe $A$, la création d'une instance de $B$ commence obligatoirement par la création d'une instance de $A$, à laquelle s'ajoutent les spécificités de $B$ par rapport à $A$. En conséquence, toute invocation d'un constructeur de $B$ commence nécessairement par l'invocation d'un constructeur de $A$. Ce mécanisme peut être implicite (assuré automatiquement par Java), sauf s'il est pris en charge explicitement par le programmeur.

Ainsi, si un programmeur, en écrivant un constructeur pour $B$, n'invoque pas lui-même explicitement l'un des constructeurs de $A$, alors Java se charge de le faire à sa place, en rajoutant implicitement un appel au constructeur implicite de $A$ (sans paramètres).

Sinon, un constructeur de la super classe peut être appelé explicitement au moyen de l'appel réservé `super(. . .)`. Cet appel ne peut se faire que dans un constructeur de la classe dérivée, et de plus l'appel à `super(. . .)` doit être la première instruction du constructeur de la classe dérivée. On peut bien sûr invoquer `super(. . .)` avec des paramètres si l'on souhaite invoquer un constructeur avec paramètres de la super classe.

**Attention.** L'invocation explicite d'un constructeur de la super classe invalide son invocation implicite.

**Attention.** Le constructeur de la super-classe implicitement invoqué par Java est un constructeur sans paramètre. Si la super-classe n'en possède pas, il n'est pas possible de compiler une telle classe dérivée !

**Exemple.** Dans le code suivant, la classe $B$ ne peut pas être compilée.

```
classe A {
    entier x, y
    Constructeur(entier x, entier y)
        moi.x ← x
        moi.y ← y
    fin
}
```

```
classe B hérite de A {
    entier z
}
```

En effet, $A$ déclare explicitement un constructeur, et ne possède donc plus de constructeur implicite. Et comme elle ne déclare pas elle même de constructeur sans paramètre, elle n'en possède aucun.

En revanche, $B$ ne déclare aucun constructeur. Elle possède donc un constructeur implicite, sans paramètre. Ce dernier invoque automatiquement un constructeur sans paramètre de $A$. Comme $A$ n'en possède pas, $B$ ne peut pas être compilée.

# 5 L'héritage pour regrouper des objets

L'un des avantages fondamentaux offerts par l'héritage est de pouvoir regrouper dans des tableaux (ou toute autre structure collective) d'un type donné, des objets qui sont soit de ce type donné, soit d'un type dérivé.

**Exemple.** Imaginons que l'on stocke dans un tableau une série de références, certaines à des horaires, et d'autres à des horaires précis. Il suffit de déclarer le tableau suivant :

```
Horaire[] tabHoraire ;
tabHoraire ← créer Horaire[20] ;
```

pour pouvoir stocker les références à 20 horaires, *qu'ils soient précis ou pas.*

Cela est rendu possible par le fait qu'un horaire précis *est un* horaire, comme indiqué par la sémantique de la relation d'héritage. On parle de transtypage ascendant (nous y reviendrons dans le chapitre 11 consacré au polymorphisme).

Ainsi, l'instruction

```
tabHoraire[5] ← créer HorairePrecis(8,8,8)
```

est parfaitement valide et ne provoquerait pas d'erreur de compilation.

**N.B.** En généralisant, on constate qu'on peut stocker tout objet dans un tableau de Object.