# Contenu du cours

1. Introduction à la POO
2. Une **classe** : définition de nouveaux objets
3. **Instanciation** et utilisation d'objets
4. Création des objets : les **constructeurs**
5. **Références**, visibilité des variables
6. **Encapsulation** et masquage des données
7. **Statique**, ou d'instance ?
8. **Héritage**
9. **Polymorphisme**
10. **Classes abstraites** et **interfaces**
11. **Introduction aux types génériques**
12. **Exceptions** en java
13. *Compléments syntaxiques*

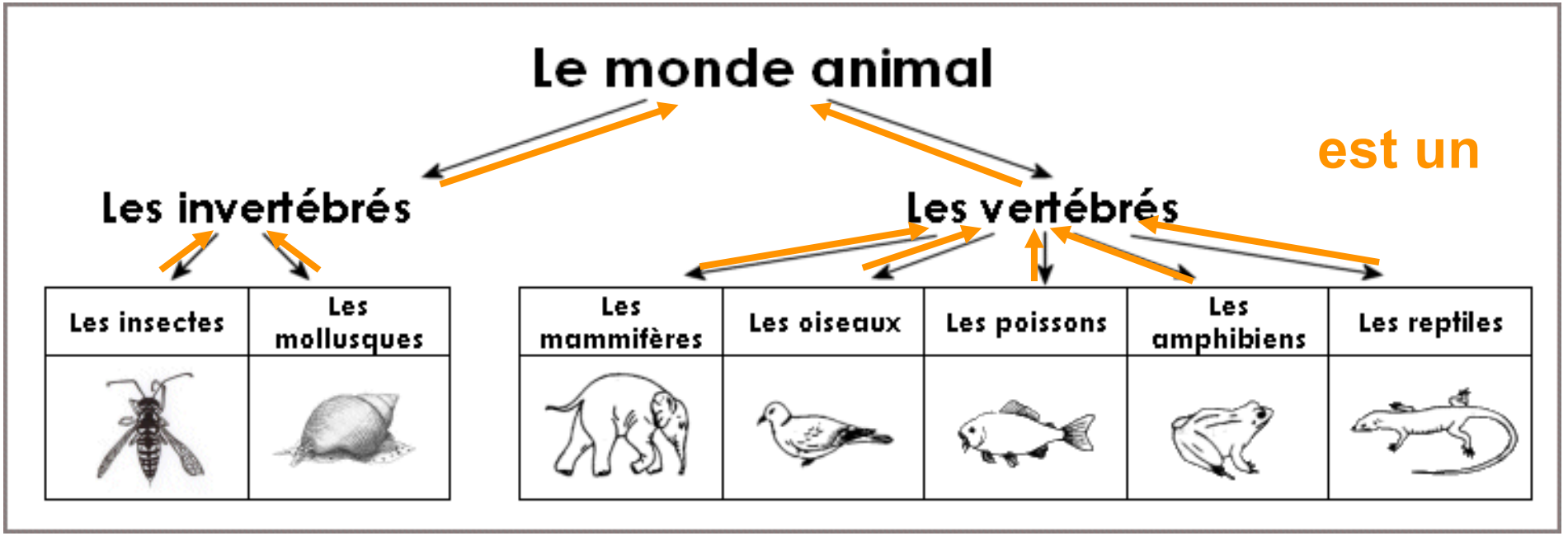
# Héritage en POO

# Le concept d'héritage en POO

- Héritage = concept fondamental de la POO
- Principe
  - Définir un nouveau type en spécialisant un type existant
  - Le nouveau type hérite des attributs et méthodes du type existant
  - Il redéfinit ce qui change par rapport à l'existant
- Quel intérêt ?
  - Coder rapidement un nouvel objet
  - Clarté conceptuelle
    - L'accent est mis sur ce qui change
  - Polymorphisme : l'objet qui hérite possède plusieurs types
    - Le sien, mais aussi celui de ses ancêtres
    - On peut « ranger ensemble » ancêtres et héritiers

# Mécanisme de l'héritage = sous-classement

- Sous-classement = base de la classification
- Exemple : la classification du règne animal



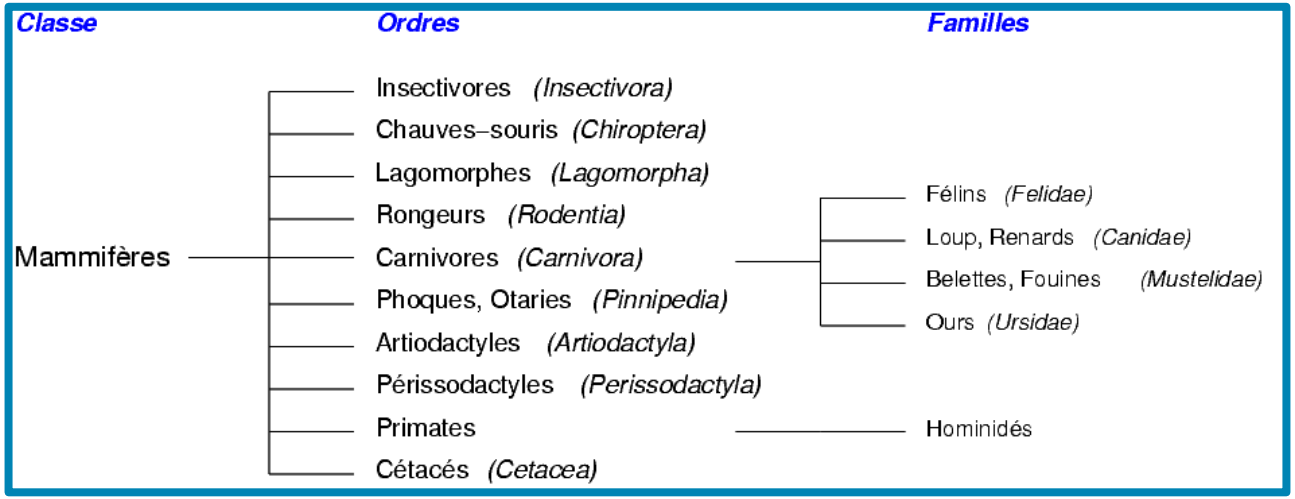

La relation d'héritage est une relation « est un »

On inversera en POO le sens des flèches

# Relation d'héritage : sémantique

Si on ne peut pas dire « est un(e) », il n'y a pas héritage

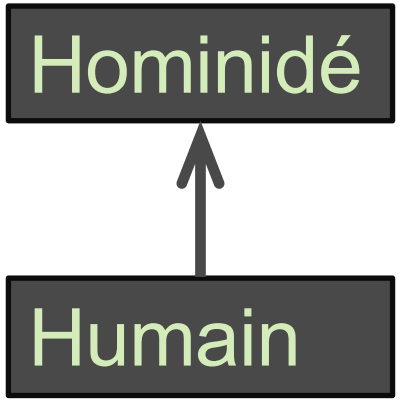
- **Exemple : les mammifères**




  - **Un humain est un hominidé (non représenté sur le schéma)**
  - **Un hominidé est un primate**
  - **Un primate est un mammifère**
- **Relation transitive**
  - **Un humain est un mammifère (par transitivité)**
- **Relation non-symétrique**
  - **Tous les mammifères ne sont pas des primates**

# Héritage : vocabulaire, syntaxe UML

- Voyons Humain et Hominidé comme des classes
- On dit que
  - Humain hérite de Hominidé
  - Humain est une sous-classe de Hominidé
  - Humain est une classe dérivée de Hominidé
  - Hominidé est la super-classe de Humain
- Notation UML
  - Flèche dirigée de la sous-classe vers la super-classe



# Héritage : syntaxe PDL++ et Java

- PDL++ : mot-clé **hérite de**

```
classe Hominide {
    …
}
```

```
classe Humain hérite de Hominidé {
    …
}
```

- Java : mot-clé **extends**

```
class Hominide {
    …
}
```

```
class Humain extends Hominidé {
    …
}
```

# Exemple : la classe Horaire en Java

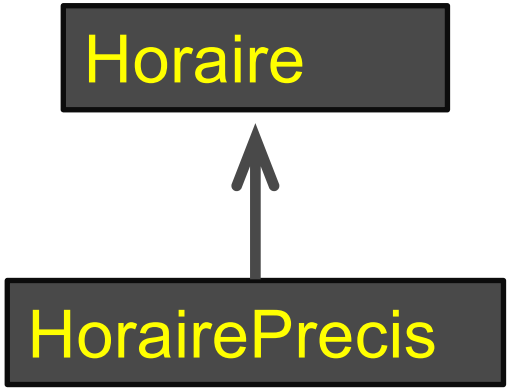
CODE JAVA

- **Code Java de la classe Horaire**

```
class Horaire {
     int h, m, s; // heures, minutes et secondes

    void setHoraire(int h, int m, int s) {
       if (h >= 0 && h < 24 && m >= 0 && m < 60 && s >= 0 && s < 60)  {
            this.h = h;
            this.m = m;
            this.s = s;
       }
   }
}
```

- **On veut un horaire plus précis, avec millisecondes**
- **On va procéder par héritage**
  - **Un horaire précis est un horaire**



# Classe HorairePrecis héritant de Horaire (version 1, JAVA)

CODE JAVA

- **Définition par héritage de HorairePrecis**

```
class HorairePrecis extends  Horaire {
      int ms; // les millisecondes

     void setHoraire(int h, int m, int s, int ms) {
         setHoraire(h, m, s);
         this.ms = ms;
     }
}
```

On ne définit que ms.
Les autres attributs (h, m, s) sont *hérités* (ils sont donc présents quand même, mais par héritage).

On n'a plus qu'à préciser ce qui se passe pour ms, qui est un nouvel attribut.

Invocation de la méthode setHoraire(…) *héritée* de la super-classe. N.B. Ce n'est pas un appel récursif (comparer le nombre de paramètres).

Tou(te)s les attributs et méthodes (non privés) de Horaire sont connus et utilisables dans HorairePrecis

# Classe HorairePrecis héritant de Horaire (version 2)

CODE JAVA

- **Redéfinition de méthodes**

Redéfinir une méthode : dans une sous-classe, on définit une méthode portant le même nom et la même signature que dans la super-classe.

```
class HorairePrecis extends Horaire {
    int ms; // les millisecondes

    void setHoraire(int h, int m, int s) {
        super.setHoraire(h, m, s);
        this.ms = 0;
    }

    void setHoraire(int h, int m, int s, int ms) {
        setHoraire(h, m, s);
        moi.ms =  ms;
    }
}
```

Redéfinition de la méthode setHoraire(…) : elle a la même signature que la méthode *héritée* de la super-classe.

Invocation de la méthode setHoraire(…) héritée.

On préfixe par le mot clé super pour indiquer qu'on veut la méthode de la super-classe. Sans cela, ce serait un appel récursif à la méthode redéfinie.

Pas une redéfinition mais une surcharge : même nom qu'une méthode existante, mais signature différente.

Coder, puis encapsuler : OK

# Redéfinition, super et super(…)

Pourquoi redéfinir une méthode ?

Permet
- d'adapter si besoin son comportement pour une sous-classe
- tout en maintenant un appel identique à la super-classe (cf. polymorphisme)

- **Mot-clé super utilisé comme un nom d'instance**
  - **super est une référence à la super-classe**
  - **Permet de désigner les attributs ou invoquer les méthodes de la super-classe**
  - **Exemple : `super.setHoraire(h, m, s)`**
- **Mot-clé super(…) utilisé comme appel de méthode**
  - **Permet d'invoquer un constructeur de la super-classe**
  - **cf. Héritage et chaînage des constructeurs**

Coder un exemple SetHorairePrecis

# Classe inhéritables : le mot-cle final

- Rappel : une variable final est une constante
  - Sémantique : la variable n'est pas modifiable
  - Dit autrement : sa valeur ne peut pas être *redéfinie*
- Une classe aussi, peut être déclarée final
  - Sémantique : la classe n'est pas héritable
  - Dit autrement : son comportement ne peut pas être *redéfini*
  - Mot-clé final identique pour Java et PDL++
- Exemple
  - `public final class A { … }`
  - `public class B extends A { … }` : échec compilation

```
error: cannot inherit from final  A
```

- Une méthode aussi peut être final
  - Sémantique : impossible de la redéfinir

Tenter d'hériter d'une classe Horaire final : KO

# Héritage et modificateurs de visibilité

- **Rappel : les attributs et les méthodes de la super-classe sont hérité(e)s par la sous-classe**
  - **Ils/elles sont directement utilisables dans la sous-classe**

Est-ce toujours vrai ?

Non, ce n'est plus vrai si les attributs/méthodes sont privé(e)s

Utilisation déconseillée. Préférer une utilisation maîtrisée de l'encapsulation.

Alors, comment masquer un attribut/une méthode, mais qu'il/elle reste héritable ?

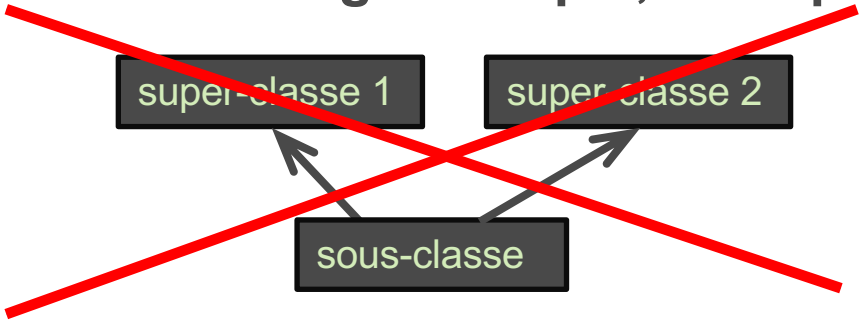
On peut le/la déclarer protected, mais il/elle devient alors visible dans tout le package

- **Modificateur protected (java) / protégé (PDL++)**
  - **Sémantique : *défaut*, sauf pour les sous-classes**

Masquer h, m, s : OK si encapsulation correcte

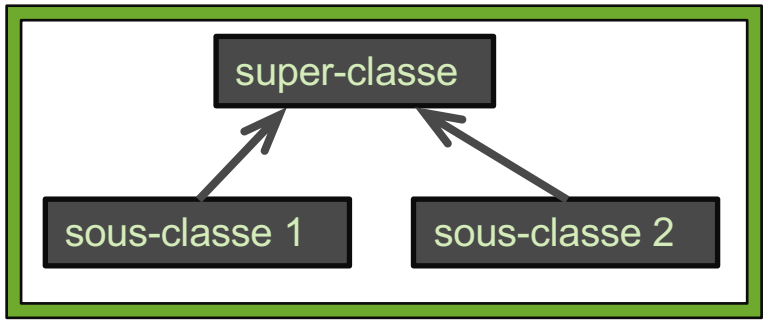
# Héritage en série : OK, héritage multiple : KO

- **Pas d'héritage multiple en java**
  - **Une sous-classe java ne possède qu'une seule superclasse**
  - **C++ autorise l'héritage multiple, mais pas java**




INTERDIT en Java

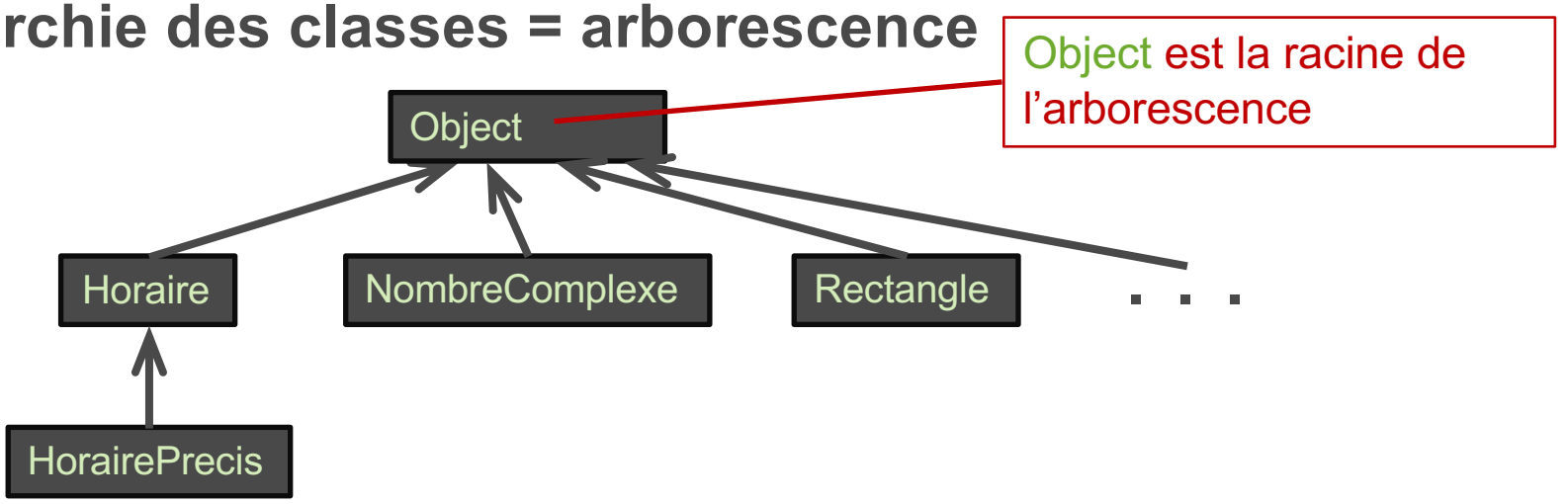
- **Héritage en série**
  - **Dans l'autre sens pas de soucis : une super-classe peut avoir plusieurs sous-classes**




AUTORISÉ

# Hiérarchie des classes

- Toute classe java possède une super-classe
  - Même si elle n'est pas spécifiée explicitement par **extends**
  - La super-classe java par défaut est la classe **Object**
- Par transitivité, toute classe dérive de **Object**
- Hiérarchie des classes = arborescence




- Classe **Object** : voir son contenu dans l'API java
  - Définit la méthode **toString()**
  - **toString()** est à redéfinir par chaque sous-classe

NombreComplexe@677327b6

# Héritage : comment ça fonctionne ?

Toute instance d'une sous-classe contient en elle une instance de sa super-classe.

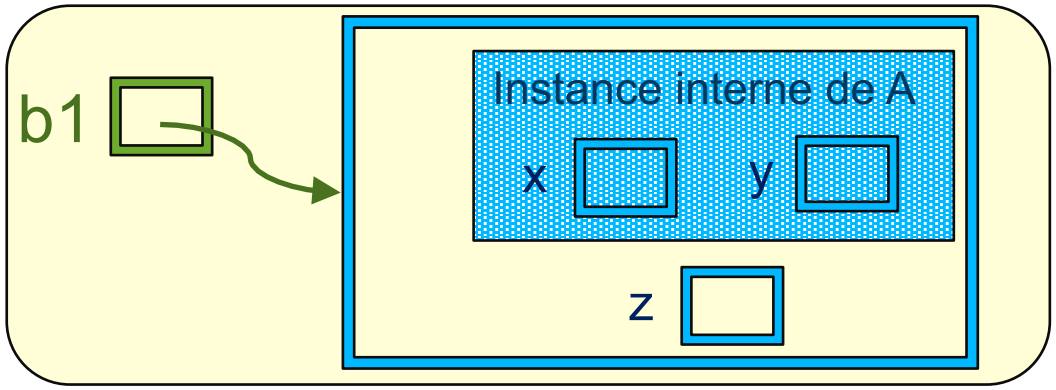
- **Ainsi la sous-classe contient**
  - **Ses propres attributs et méthodes**
  - **Mais aussi les attributs et méthodes de sa super-classe**

```
class  A {
      int x, y;
}
```

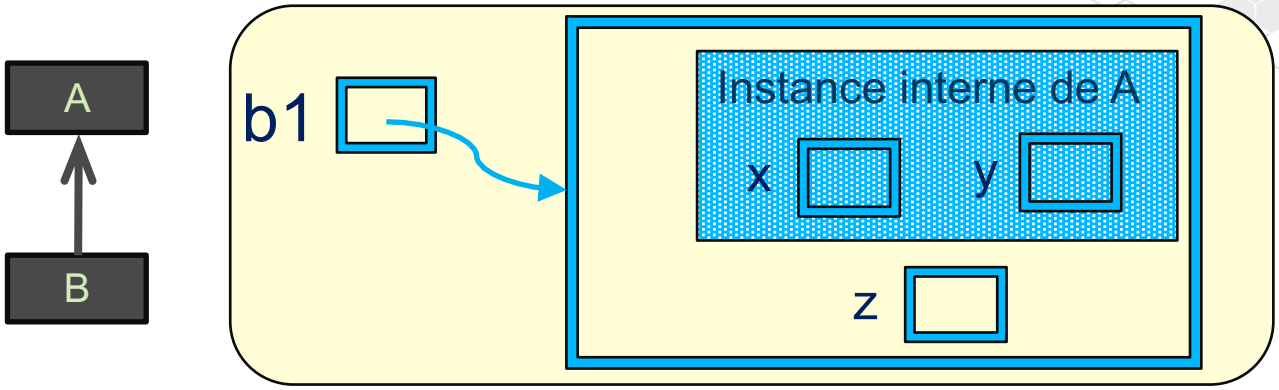
```
class B extends A {
      int z;
}
```

```
… main(…) {
      B b1 = new B();
}
```

*(situation mémoire)*



# Chaînage des constructeurs




- **Création d'une instance de B**
  - **Commence par la création d'une instance de A**
  - **Puis s'ajoutent les spécificités de B**
- **Au niveau des constructeurs**

  Chaque constructeur de B débute nécessairement par l'invocation implicite ou explicite d'un constructeur de A

  - **Invocation implicite**
    - **Insertion automatique par java (en 1ère instruction) de super()**
  - **Invocation explicite**
    - **A réaliser par le programmeur avec super(…) (1ère instruction)**

La rédaction d'un constructeur explicite invalide l'ajout automatique de l'appel à **super()** par java.

# Constructeur oublié ==> risque d'erreur

Le constructeur de la super-classe implicitement invoqué est un constructeur sans paramètre.

- **Exemple de classe non compilable**

```
class A {
    int x, y;

    A(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

```
class B extends A {
    int z;
}
```

Coder cet exemple

*Pas de constructeur sans paramètre dans A ==> appel impossible*

- B n'a pas de constructeur
- Java ajoute implicitement le constructeur suivant :

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

- **La classe B ne peut pas être compilée**

```
B.java:1: error: constructor A in class A cannot be applied to given types;
public class B extends A {
       ^
  required: int,int
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```