Informatique / Programmation

Programmation orientée objet avec Java

12 : Sous-classes, héritage et polymorphisme

Jacques Bapst

jacques.bapst@hefr.ch

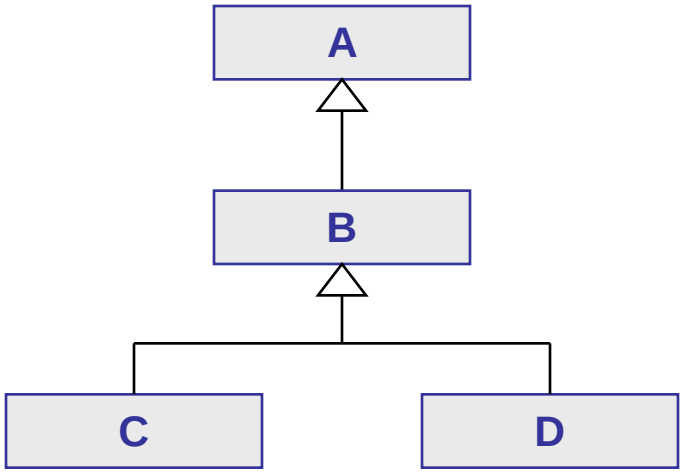
Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Sous-classe et héritage

- L'**héritage** est une **propriété essentielle** de la programmation orientée objet.
- Ce mécanisme permet d'ajouter des fonctionnalités à une classe (une **spécialisation**) en créant une **sous-classe** qui hérite des propriétés de la classe parente et à laquelle on peut ajouter des propriétés nouvelles.
- La classe qui hérite est appelée **sous-classe** (ou **classe dérivée**) de la **classe parente** (qui est également appelée **super-classe**).
- L'héritage permet à une sous-classe d'**étendre les propriétés** de la classe parente tout en héritant des champs (attributs) et des méthodes (comportement) de cette classe parente.
- En *Java*, une classe ne peut hériter que d'une seule classe parente. On parle dans ce cas d'**héritage simple** (par opposition à l'héritage multiple qui permet à une classe d'hériter de plusieurs classes parentes).

Arborescence de classes

- L'héritage induit une relation arborescente entre les classes.



En notation UML, le symbole 

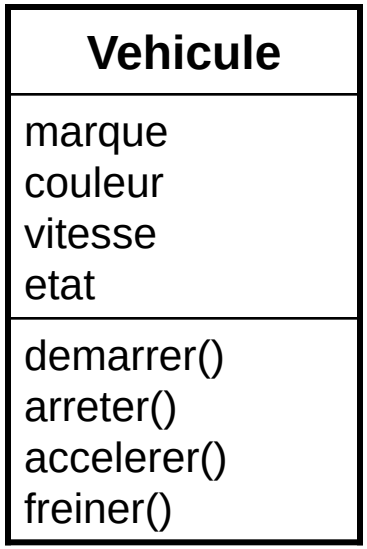
signifie
"*est une sous-classe de*"
"*hérite de*"

- La classe **A** est la classe parente (super-classe) de **B**
- La classe **B** est la classe parente de **C** et **D**
- La classe **B** est une sous-classe de **A**
- Les classes **C** et **D** sont des sous-classes de **B**

B joue plusieurs rôles

Exemple : classe `Vehicule`

- Pour illustrer le concept, prenons l'exemple d'une classe `Vehicule` permettant de décrire (très sommairement) les propriétés et les comportements d'un véhicule du monde réel.
- Avec une modélisation très simpliste, la classe pourrait être représentée de la manière suivante :

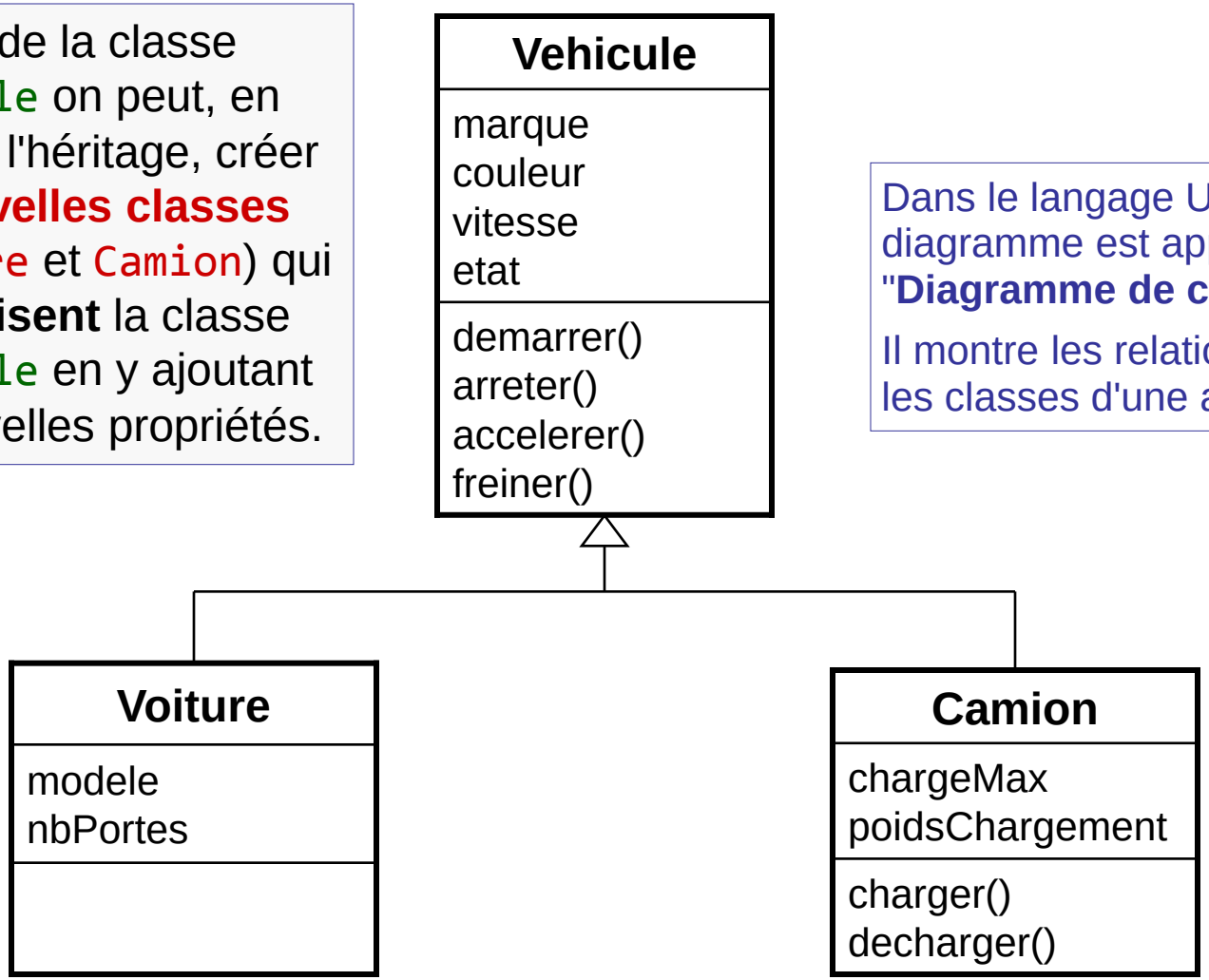


Implémentation classe Vehicule

```
public class Vehicule {

  private String marque;
  private String couleur;
  private double vitesse;      // Vitesse actuelle
  private int    etat;         // 0:arrêt, 1:marche, …

  public Vehicule(String marque, String couleur) {
    this.marque  = marque;
    this.couleur = couleur;
    vitesse      = 0.0;
    etat         = 0;
  }

  public void demarrer() { etat = 1; }

  public void arreter()  { if (vitesse == 0) etat = 0; }

  public void accelerer() {
    if (etat == 1) vitesse += 5.0;
  }

  public void freiner() {
    if (vitesse >= 5.0) vitesse -= 5.0;
    else                vitesse  = 0.0;
  }
}
```

Sous-classes de `Vehicule`

A partir de la classe `Vehicule` on peut, en utilisant l'héritage, créer de **nouvelles classes** (`Voiture` et `Camion`) qui **spécialisent** la classe `Vehicule` en y ajoutant de nouvelles propriétés.



Dans le langage UML, un tel diagramme est appelé "**Diagramme de classes**".

Il montre les relations entre les classes d'une application

Sous-classes `Voiture` et `Camion`

- La classe `Voiture` est une sous-classe de `Vehicule`.
  Elle hérite de tous les attributs et méthodes de `Vehicule` (`marque`, `couleur`, ..., `freiner()`) en y ajoutant deux nouveaux attributs (`modele` et `nbPortes`).
- La classe `Camion` est également une sous-classe de `Vehicule` et hérite de tous ses attributs et méthodes. La classe `Camion` étend la classe parente (`Vehicule`) en y ajoutant deux nouveaux attributs (`chargeMax` et `poidsChargement`) ainsi que deux nouvelles méthodes (`charger()` et `decharger()`) .
- Dans les sous-classes `Voiture` et `Camion`, on peut utiliser les attributs et les méthodes héritées (par exemple `couleur` ou `freiner()`) comme si ces membres avaient été déclarés dans chacune des sous-classes (sous réserve, naturellement, que les **droits d'accès** le permettent).

Déclaration de sous-classe

- La déclaration d'une sous-classe s'effectue en utilisant le mot-clé `extends` suivi du nom de la classe parente :

```
public class Voiture extends Vehicule {

  private String modele;                  // Nouveau champ
  private int    nbPortes;                // Nouveau champ

  public Voiture(String marque,           // Constructeur
                 String modele,
                 String couleur,
                 int    nbPortes) {

    super(marque, couleur);   // Appelle le constructeur de la classe parente

    this.modele   = modele;
    this.nbPortes = nbPortes;
  }
}
```

Relations entre classes [1]

- L'héritage entre une sous-classe et sa classe parente est caractérisé par une **relation** de type

  *"**Est un...**"* (*"Is a..."*)
  ou *"**Est une sorte de...**"* (*"Is kind of..."*)

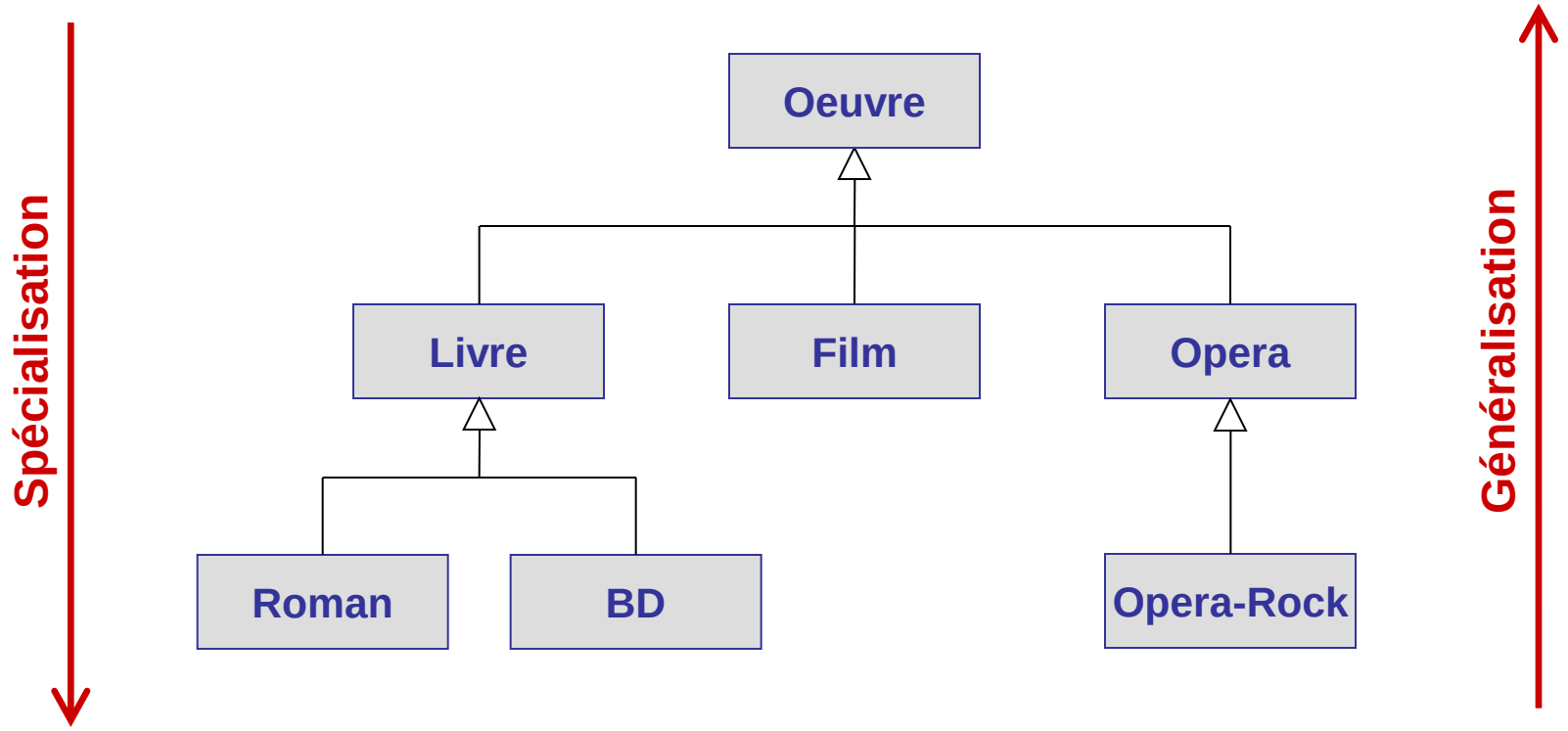
  - Une voiture **est un** véhicule
  - Un camion **est un** véhicule

  Attention *: l'inverse n'est pas forcément vrai !*

- Une sous-classe crée une **spécialisation** de la classe parente. A l'inverse, on parle de **généralisation** lorsqu'on passe des sous-classes à leur classe parente.
- Point à retenir lors de la conception d'une application :

"Est un..." ⇨ Héritage

Généralisation / Spécialisation



Relations entre classes [2]

- Il existe une autre relation importante qui peut exister entre deux classes. Il s'agit de la relation de **composition** (ou **agrégation**) qui est caractérisée par une **relation** de type

  *"**A un...**"* *("Has a...")*
  ou *"**Possède un...**"*
  ou *"**Est composé de...**"*
  ou *"**Contient...**"*

  - Une voiture **possède un** moteur
  - Une voiture **a un** propriétaire

"A un..." ⇨ Composition
⇨ Agrégation

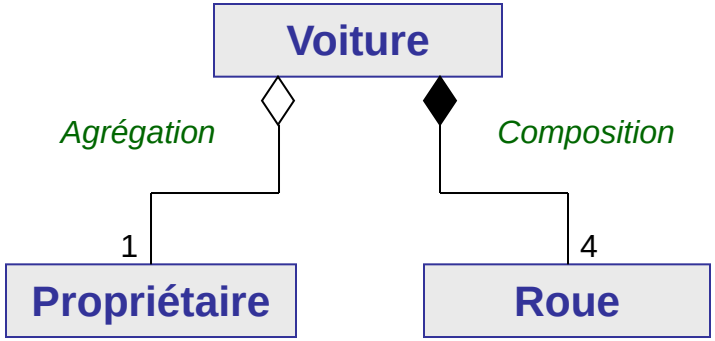
- En *Java*, les relations de composition sont réalisées en créant dans la classe *"contenant"* une **référence vers** un objet de la classe *"contenu"*.

```
public class Voiture extends Vehicule {

  private Personne  propriétaire;
  private FuelMotor moteur;
  . . .
```

Relations entre classes [3]

Complément

- En **notation UML**, la relation *"**A un**"*, *"**Possède un**"*, … est représentée de la manière suivante :



- La distinction sémantique entre **composition** et **agrégation** ne se traduit pas (en Java) par des différences d'implémentation.

```
public class Voiture extends Vehicule {
  . . .
   private Personne  leProprietaire;
   private Roue[]    lesRoues;
  . . .
}
```

Généralisation

- La relation d'héritage *("Est un...")* permet de traiter les objets des sous-classes comme s'ils étaient des objets de leur classe parente (par généralisation).

Important !

```
Camion   c1 = new Camion(…);
Vehicule v1 = c1;            // Ok, un camion est un véhicule
Camion   c2 = v1;            // ERREUR, un véhicule n'est pas forcément un camion !
Camion   c3 = (Camion)v1;    // Ok, v1 référence effectivement un camion
```

- Si nécessaire, le système effectue donc une **conversion élargissante** (automatique) de la sous-classe vers la classe parente *(Upcasting)*.
- Une **conversion explicite** (transtypage, *casting*) d'un objet de la classe parente vers un objet de la sous-classe *(Downcasting)* est possible si l'instance à convertir référence effectivement (au moment de l'exécution) un objet de la sous-classe considérée (sinon, il y aura une erreur `ClassCastException` à l'exécution).

Opérateur `instanceof`

- L'opérateur **`instanceof`** permet de tester (à l'exécution) l'appartenance d'un objet à une classe (ou une interface) donnée.
- Dans l'exemple précédent, on aurait pu écrire :

```
if (v1 instanceof Camion) {
  c3 = (Camion)v1;
}
else {
  ...  // v1 ne référence pas un Camion
}
```

Un tel test permet d'éviter une erreur fatale (à l'exécution) si la variable `v1` ne référence pas un objet de type `Camion`.

Remarque 1 : `(v1 instanceof Vehicule)` est également vrai !

Remarque 2 : L'*upcasting* et le *downcasting* n'engendrent aucune opération ni changement en mémoire (c'est une autre "vue" de l'objet)

Classe `Object`

- En *Java*, chaque classe que l'on crée possède une classe parente.
- Si l'on ne définit pas explicitement une classe parente (avec `extends`), la super-classe par défaut est `Object` (qui est déclarée dans le paquetage `java.lang`).
- La classe `Object` est donc l'ancêtre de toutes les classes *Java* (c'est la racine unique de l'arbre des classes).
- La classe `Object` est la seule classe *Java* qui ne possède pas de classe parente.
- Toutes les classes héritent donc des méthodes de la classe `Object` (par exemple `toString()`, `equals()`, `finalize()`, etc).
- La classe `Object` constitue la généralisation ultime :

Tous les objets sont des `Object` *!*

Chaînage des constructeurs

- Un constructeur d'une sous-classe peut faire **appel à un constructeur de la classe parente** en utilisant le mot réservé `super` selon la syntaxe suivante :

  `super(`*Expr1, Expr2, ...*`);`

- Si un constructeur d'une sous-classe invoque explicitement un constructeur de la classe parente, l'instruction `super(...)` doit être **la première instruction du constructeur**.
- Si l'on ne fait pas explicitement appel à un constructeur de la classe parente, une **invocation du constructeur par défaut** de la super-classe (constructeur sans paramètre) sera **automatiquement ajoutée** (comme première instruction). Si un tel constructeur n'existe pas dans la classe parente, une erreur sera générée à la compilation.
- Le langage garantit ainsi que tous les éléments des classes parentes aient été élaborés avant la création d'un objet de la classe considérée.

Masquage des champs

Complément

- Si une sous-classe définit un champ avec le même nom qu'un champ de sa classe parente (une pratique **à éviter**), alors le champ de la super-classe est **masqué** dans le corps de la sous-classe.
- Dans ce cas, on peut, dans le corps de la sous-classe accéder au champ de la classe parente en utilisant le mot-clé `super` suivi d'un point et du nom du champ.
- Si les classes `A` et `B` définissent toutes deux un champ `x` et que `B` est une sous-classe de `A` alors, dans les méthodes de `B`, on peut utiliser :

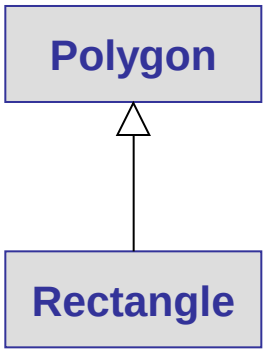
`x`	Champ `x` de la classe `B`
`this.x`	Champ `x` de la classe `B`
`super.x`	Champ `x` de la classe `A`
`((A)this).x`	Champ `x` de la classe `A` (par transtypage)

La notation `super.super.x` n'est pas autorisée, seul le casting (`((A)c).x`) permet d'accéder à un champ masqué d'une classe ancêtre.

- Les champs statiques peuvent également être masqués mais ils restent accessibles en les préfixant avec le **nom de la classe**.

Redéfinition des méthodes [1]

- Lorsqu'une classe définit une **méthode d'instance** en utilisant la même signature (même nom, type de retour [év. une sous-classe], paramètres, etc.) qu'une méthode de sa classe parente, cette méthode **redéfinit** (***overrides***) la méthode de sa super-classe.
- Par exemple :




```
public class Polygon {
  . . .
  public double area() {
    . . .   // Calcul de la surface d'un polygone quelconque
  }
  . . .
}
```

```
public class Rectangle extends Polygon {
  . . .
  public double area() {
    . . .   // Calcul de la surface du rectangle (longueur*largeur)
  }
  . . .
}
```

Redéfinition des méthodes [2]

- Les classes `Polygone` et `Rectangle` définissent toutes deux une méthode `area()` avec une signature identique.
- La méthode invoquée par un appel `obj.area()` dépendra du type de l'objet référencé par la variable `obj` lors de l'exécution de cette instruction.
- C'est **toujours la méthode associée au type effectif de l'objet référencé** qui est exécutée, même si l'objet est enregistré dans une variable déclarée avec le type d'une classe parente :

```
Rectangle r = new Rectangle(2.6, 5.3);

Polygon   p = r;          // Conversion élargissante (Upcasting)

double s = p.area();      // Invoque r.area() et non pas area() de la
                          // classe Polygon car la variable p référence
                          // un objet de type Rectangle
```

Redéfinition des méthodes [3]

- Dans la sous-classe `Rectangle`, il est possible d'invoquer la méthode `area()` de la classe parente `Polygon` en utilisant la notation suivante :

```
super.area()
```

Remarque : *La notation `super.super.f()` n'est pas autorisée et il est impossible d'accéder à une méthode masquée d'une classe ancêtre (autre que la classe parente).*

- A l'extérieur de la classe de déclaration, il n'est, par contre, pas possible pour un objet de type `Rectangle`, d'invoquer la méthode `area` de la classe `Polygon`.
- Attention à bien distinguer (et ne pas confondre) les notions de **surcharge** *(overloading)* et de **redéfinition** *(overriding)* de méthodes.

Redéfinition des méthodes [4]

- L'annotation `@Override` peut être utilisée pour indiquer explicitement l'intention de redéfinir une méthode.
- Bien qu'étant facultative, elle apporte deux avantages :
  - Si on se trompe en écrivant le nom de la méthode lors de la redéfinition, le compilateur peut alors informer le programmeur de son erreur.
  - La redéfinition étant explicite, le code est plus clair.

```
@Override
public double area() { … }
```

Complément

Redéfinition des méthodes [5]

Complément

- Les **méthodes statiques** (méthodes de classe) peuvent être masquées, dans des sous-classes, par des méthodes statiques possédant le même nom (*** une pratique **à éviter** ***).
- Ces méthodes masquées restent cependant accessibles en les préfixant avec le nom de la classe dans laquelle elles sont définies (comme il est recommandé de le faire avec toutes les méthodes statiques).
- On ne peut donc pas spécialiser une méthode statique dans une sous-classe (comme les méthodes statiques ne sont jamais associées à un objet, le polymorphisme de s'applique pas).
- En outre, une **méthode statique ne peut pas masquer une méthode d'instance** d'une super-classe (erreur à la compilation).

Redéfinition des méthodes [6]

Complément

- Si une méthode est redéfinie dans une sous-classe, le **type de retour doit être identique** à celui de la méthode correspondante de la classe parente.
- En redéfinissant une méthode, il est possible d'étendre sa **zone de visibilité** (`private` → *package* → `protected` → `public`) mais non de la restreindre.
- Une méthode redéfinie ne peut pas générer plus d'**exceptions** contrôlées que celles qui sont déclarées dans la méthode de la classe parente (mais elle peut en déclarer moins).
- Éviter de :
  - surcharger une méthode qui est redéfinie
  - redéfinir une partie des méthodes surchargées

  car cela provoque des risques de confusion et crée des pièges lors des modifications ultérieures des classes.

Modificateur `final`

- Dans la **déclaration d'une classe**, le modificateur **`final`** indique que la classe ne peut pas être sous-classée (on ne peut pas créer de classes dérivées).
- Dans la **déclaration d'un champ**, le modificateur **`final`** indique que la valeur du champ ne peut pas être modifiée après l'affectation initiale (dans la déclaration ou dans le constructeur). Permet de définir des valeurs constantes.
- Dans la **déclaration d'une méthode**, le modificateur **`final`** indique que la méthode ne peut pas être redéfinie dans une sous-classe.
- Dans la **déclaration des paramètres d'une méthode**, le modificateur **`final`** indique que la valeur de ces paramètres ne peut pas être modifiée (il s'agit de paramètres *d'entrée* de la méthode).

Remarque : L'utilisation du modificateur `final` permet en outre au compilateur d'effectuer certaines optimisations : mise en ligne de méthodes *(inlining)*, suppression de la recherche dynamique *(late binding)*, etc.

Modificateur `protected`

- Le modificateur **`protected`** appliqué aux membres (champs ou méthodes) d'une classe indique que ces champs ne sont accessibles que dans la classe de définition, dans les classes du même paquetage et dans les sous-classes de cette classe (indépendamment du paquetage).
- Il s'agit d'un accès plus restrictif que `public` mais - contrairement à ce que son nom pourrait laisser croire - **moins restrictif que l'accès par défaut** (*package*).
- Le modificateur `protected` devrait être utilisé avec les champs et les méthodes qui ne sont pas requis par les utilisateurs de la classe mais qui pourraient s'avérer utiles à la création de sous-classes dans d'autres paquetages.

Modificateur `private`

- Le modificateur **`private`**, appliqué aux membres (champs ou méthodes) d'une classe, indique que ces champs ne sont accessibles que dans la classe de définition.
- Même s'il ne sont pas accessibles dans les sous-classes, les champs privés sont **malgré tout hérités** dans les sous-classes (une zone mémoire leur est allouée).
- Il s'agit de l'accès **le plus restrictif**.
- Le modificateur `private` devrait être utilisé avec les champs et les méthodes qui ne sont utilisés qu'au sein de la classe de définition et qui devraient être cachés partout ailleurs.

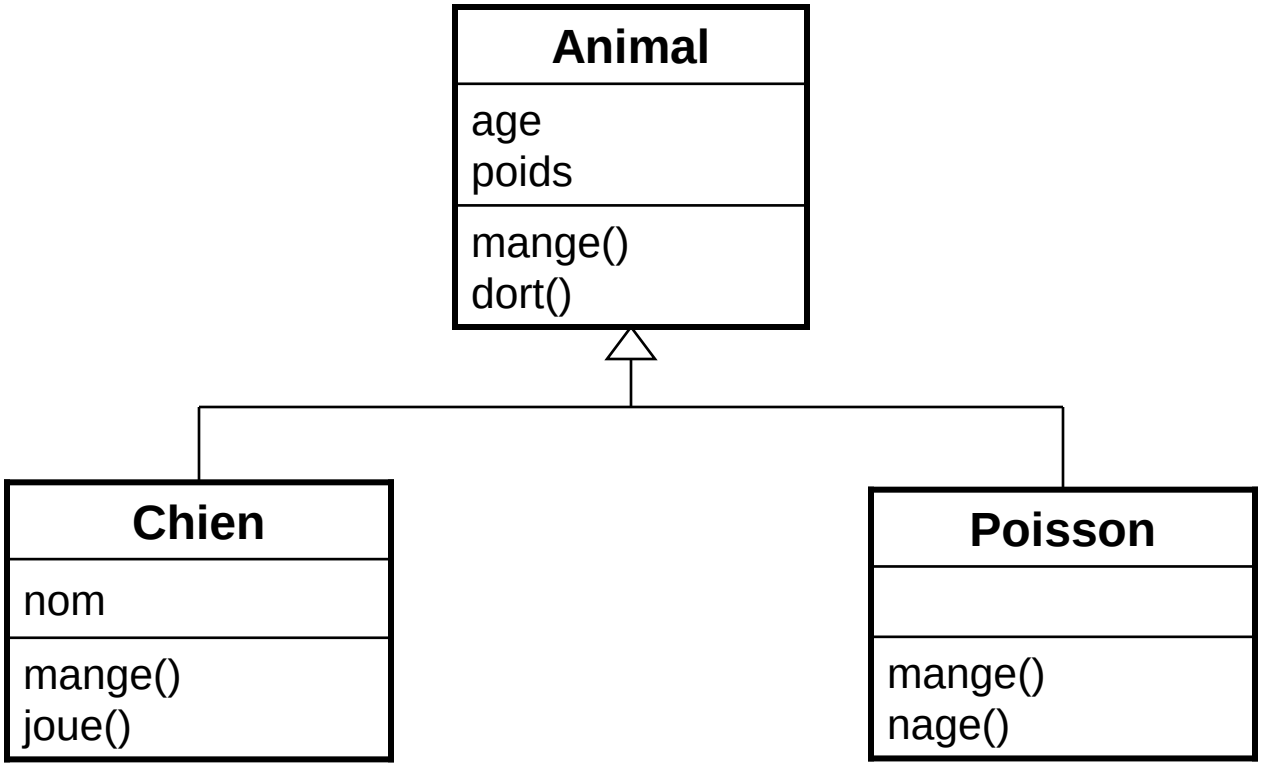
Conseil : D'une manière générale il vaut mieux **commencer par un accès restrictif** aux membres d'une classe (⇨ `private`).

Si nécessaire, il est toujours possible de relâcher les restrictions dans des versions ultérieures de la classe.

Un relâchement des restrictions préserve la **compatibilité ascendante** (ce qui n'est pas le cas si l'on restreint les droits d'accès).

Polymorphisme [1]

- Si l'on considère le diagramme de classes suivant :



Polymorphisme [2]

- On peut déduire du diagramme de classes les considérations suivantes :
  - `Chien` et `Poisson` sont des **sous-classes** d'`Animal`
  - `Animal` est la **classe parente** de `Chien` et de `Poisson`
  - `Chien` **hérite** des membres d'`Animal` (`age`, `poids`, `dort()`)
  - `Chien` **ajoute** le champ `nom`
  - `Chien` **ajoute** la méthode `joue()`
  - `Chien` **redéfinit** la méthode `mange()`
  - `Poisson` **hérite** des membres d'`Animal` (`age`, `poids`, `dort()`)
  - `Poisson` **ajoute** la méthode `nage()`
  - `Poisson` **redéfinit** la méthode `mange()`

Polymorphisme [3]

- Si l'on implémente les classes `Animal`, `Chien` et `Poisson` et que l'on écrit le code suivant :

```
Chien milou = new Chien(…);
milou.mange();
```

- La méthode `mange()` qui sera invoquée est celle qui est définie dans la sous-classe `Chien` (car `milou` est du type `Chien` et la méthode `mange()` est redéfinie pour cette sous-classe).

- Si la sous-classe `Chien` ne redéfinissait pas la méthode `mange()`, ce serait alors la méthode `mange()` d'`Animal` qui serait invoquée (la classe `Chien` en hériterait).

Polymorphisme [4]

- Comme un `Chien` est un `Animal` (relation d'héritage) on peut écrire le code suivant :

```
Animal toutou = new Chien(…);
toutou.mange();
```

- Quelle méthode `mange()` sera invoquée dans ce cas ?
- C'est toujours la méthode `mange()` définie dans la sous-classe `Chien` qui sera invoquée.
- Même si `toutou` est une référence de type `Animal`, c'est **le type de l'objet référencé qui détermine la méthode qui sera appelée** (comportement décrit sous *"Redéfinition des méthodes"*).

Polymorphisme [5]

- En *Java*, dans la plupart des situations où il y a des relations d'héritage, la détermination de la méthode à invoquer n'est pas effectuée lors de la compilation.
- C'est seulement à l'exécution que la machine virtuelle déterminera la méthode à invoquer selon le type effectif de l'objet référencé à ce moment là.
- Ce mécanisme s'appelle *"**Recherche dynamique de méthode**"* (***Late Binding*** ou ***Dynamic Binding***).
- Ce mécanisme de recherche dynamique (durant l'exécution de l'application) sert de base à la mise en oeuvre de la propriété appelée **polymorphisme**.

Polymorphisme [6]

- On pourrait définir le **polymorphisme** comme la propriété permettant à un programme de **réagir de manière différenciée à l'envoi d'un même message** (invocation de méthode) en fonction des objets qui reçoivent ce message.
- Il s'agit donc d'une aptitude **d'adaptation dynamique du comportement** selon les objets en présence.
- Avec l'**encapsulation** et l'**héritage**, le **polymorphisme** est une des propriétés essentielles de la programmation orientée objet.

Remarque : La surcharge de méthodes peut également être considérée comme une forme de polymorphisme. Le choix de la méthode à invoquer est cependant déterminé à la compilation

Exemple de polymorphisme [1]

- L'exemple qui suit est destiné à illustrer le principe du polymorphisme. Il se base sur les classes précédemment définies (`Animal`, `Chien` et `Poisson`).
- Si l'on souhaite enregistrer et manipuler une collection d'animaux (une ménagerie) on peut créer et alimenter le tableau suivant :

```
// Déclaration et création du tableau
Animal[] menagerie = new Animal[6];

// Alimentation du tableau
menagerie[0] = new Poisson(…);
menagerie[1] = new Chien(…);
menagerie[2] = new Chien(…);
menagerie[3] = new Animal(…);
menagerie[4] = new Poisson(…);
menagerie[5] = new Chien(…);
```

Exemple de polymorphisme [2]

- Le polymorphisme nous permet d'écrire une méthode `nourrir` dont la fonction est de donner à manger à chaque animal contenu dans le tableau passé en paramètre en appelant successivement la méthode `mange()` pour chacun d'eux.

```
//-------------------------------------------------
//  Appelle la méthode  mange() pour chaque animal contenu
//  dans le tableau passé en paramètre
//-------------------------------------------------
public static void nourrir(Animal[] tabAnimaux) {

  if (tabAnimaux == null) return;

  for (int i=0; i<tabAnimaux.length; i++) {
    if (tabAnimaux[i] != null) {
      tabAnimaux[i].mange();
    }
  }
}
```

Exemple de polymorphisme [3]

- On peut ensuite appeler la méthode `nourrir()` en lui passant en paramètre la ménagerie précédemment créée :

```
nourrir(menagerie);
```

- Sur la base du contenu de `menagerie`, la méthode `nourrir()` appellera successivement :

```
mange() de la classe Poisson
mange() de la classe Chien
mange() de la classe Chien
mange() de la classe Animal
mange() de la classe Poisson
mange() de la classe Chien
```

- La méthode `nourrir()` n'a pas besoin de déterminer elle-même quelle méthode doit être appelée pour chaque animal.
- Le **polymorphisme** fera en sorte que le message `mange()` soit interprété (à l'exécution) de manière appropriée selon les objets qui le reçoivent (ainsi chaque animal mangera selon ses goûts !).

Polymorphisme – Synthèse du mécanisme

- En Java, les variables sont typées. Avec la notion d'héritage, plusieurs «types» peuvent être compatibles avec une déclaration de variable.

```
short  s;   // Ne peut rien contenir d'autre qu'un short
Object o;   // Peut contenir une référence vers un Object,
            // ou un objet d'une sous-classe
```

- Le **compilateur** s'assure que les instructions sont compatibles avec la déclaration :

```
Object o;
o = "ah";
int i=o.length();   // Refusé, car length() ∉ Object
int i=((String)o).length();   // Ok si o référence un String
```

- A **l'exécution**, Java «suit toujours la flèche» pour atteindre les méthodes :

```
Animal a;
a = new Chien();
a.mange();          // Ce sera bien la méthode redéfinie par la
                    // classe Chien qui sera exécutée
```

```
public class Animal {
  void mange() {...}
}
```

```
public class Chien extends Animal {
  void mange() {...}   // Redéfinition !
}
```