Chapitre V – Héritage

Chapitre V – Héritage

Chapitre V – Héritage

Chapitre V – Héritage

Chapitre V – Héritage

- I. Principe
- II. Un exemple
- III. Constructeurs dans une sous-classe
- IV. Polymorphisme
- V. Conversion de classe ou Transtypage
- VI. Redéfinir des méthodes
- VII. Masquage d'attributs
- VIII. La classe `Object`

Chapitre V – Héritage

- I. Principe
- II. Un exemple
- III. Constructeurs dans une sous-classe
- IV. Polymorphisme
- V. Conversion de classe ou Transtypage
- VI. Redéfinir des méthodes
- VII. Masquage d'attributs
- VIII. La classe `Object`

Chapitre V – Héritage

- I. Principe
- II. Un exemple
- III. Constructeurs dans une sous-classe
- IV. Polymorphisme
- V. Conversion de classe ou Transtypage
- VI. Redéfinir des méthodes
- VII. Masquage d'attributs
- VIII. La classe `Object`

Principe

- TOUTE CLASSE HÉRITE D'UNE ET D'UNE SEULE CLASSE APPELÉE SA *superclasse* HORMIS LA CLASSE `java.lang.Object`
- UNE CLASSE QUI N'INDIQUE PAS SA SUPERCLASSE HÉRITE AUTOMATIQUEMENT DE LA CLASSE `Object`.

Syntaxe

```
class B extends A { ....
```

On dit que B *hérite* de A ou que B est une *sous-classe* de A ou que A est la *superclasse* de B ou que B *étend* A.
B devient un ≪ sous-ensemble ≫ de A.
B peut alors utiliser tous les attributs et méthodes de sa superclasse A.
B peut aussi définir des attributs supplémentaires et ses propres méthodes qui distingueront les objets B des objets A.
Mais attention, si un attribut est déclaré `private` dans la superclasse A alors cet attribut devient un attribut de la classe B mais il n'est pas directement consultable ou modifiable dans la classe B. Il faut alors utiliser des accesseurs de la superclasse A pour manipuler ces attributs ou bien alors modifier la protection avec `protected` dans la superclasse A.

Schéma d'héritage

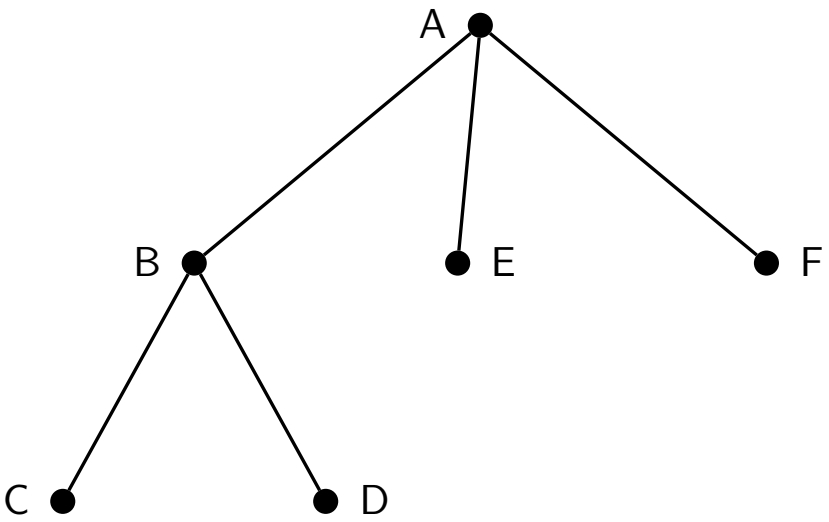
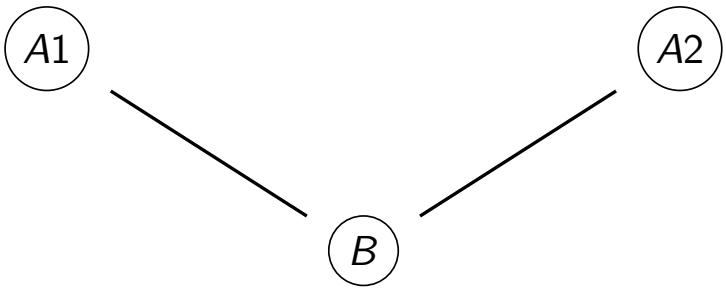


Schéma impossible



Interdiction d'héritage

```
final class ClasseFinale { ...
```

On peut interdire à une classe d'être étendue en la déclarant `final`

Un exemple

```
public class EtudiantStand{
// attributs
private String nom;
private String prenom;
private int numeroCarte;
//Constructeurs
public EtudiantStand(String nom, String prenom, int numeroCarte){
this.nom=nom;this.prenom= prenom; this.numeroCarte=numeroCarte;
}
public EtudiantStand(){}
//méthodes d'accès
public void setNom(String sonNom){
this.nom=sonNom;}
public String getNom(){
return this.nom;}
public void setPrenom(String sonPrenom){
this.nom=sonPrenom;}
public String getPrenom(){
return this.prenom;}
```

Un exemple

```
public void setNumeroCarte(int x){
this.numeroCarte=x;}
public int getNumeroCarte(){
return this.numeroCarte;}
public void afficheEtudiant(){
System.out.println("nom : " + this.getNom()) ;
System.out.println("prenom : " + this.getPrenom()) ;
System.out.println("numero de carte d etudiant : "
+ this.getNumeroCarte()) ;}}
```

(suite)

```
public class Etudiant extends EtudiantStandard {
String specialite;
int anneeDeFormation ;
// à compléter}
```

Constructeurs dans une sous-classe

Règle : TOUT CONSTRUCTEUR D'UNE CLASSE AUTRE QUE LA CLASSE `Object` FAIT APPEL SOIT À UN CONSTRUCTEUR DE SA SUPERCLASSE SOIT À UN AUTRE CONSTRUCTEUR DE SA CLASSE. Cet appel est nécessairement la première instruction de constructeur.
De plus un appel à

- un constructeur de la superclasse se fait à l'aide du mot `super`,
- un constructeur de la classe se fait à l'aide du mot `this`.

Constructeurs dans une sous-classe

Règle : TOUT CONSTRUCTEUR D'UNE CLASSE AUTRE QUE LA CLASSE `Object` FAIT APPEL SOIT À UN CONSTRUCTEUR DE SA SUPERCLASSE SOIT À UN AUTRE CONSTRUCTEUR DE SA CLASSE. Cet appel est nécessairement la première instruction de constructeur. De plus un appel à

- un constructeur de la superclasse se fait à l'aide du mot `super`,
- un constructeur de la classe se fait à l'aide du mot `this`.

Constructeurs dans une sous-classe

Règle : TOUT CONSTRUCTEUR D'UNE CLASSE AUTRE QUE LA CLASSE `Object` FAIT APPEL SOIT À UN CONSTRUCTEUR DE SA SUPERCLASSE SOIT À UN AUTRE CONSTRUCTEUR DE SA CLASSE. Cet appel est nécessairement la première instruction de constructeur. De plus un appel à

- un constructeur de la superclasse se fait à l'aide du mot `super`,
- un constructeur de la classe se fait à l'aide du mot `this`.

exemple (suite)

```
Etudiant (String nom, String prenom, int numeroCarte,
String specialite, int anneeDeFormation){
super(nom, prenom, numeroCarte);
this.specialite = specialite;
this.anneeDeFormation = anneeDeFormation}
....
Etudiant toto = new Etudiant("Smith", "John",
20113245, "Miage", 3);
....
```

si aucun constructeur de la sous-classe n'est défini, le compilateur ajoute le constructeur suivant :

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

Si un constructeur de la sous-classe oublie en première instruction l'appel à un constructeur de la super-classe, le compilateur ajoute l'instruction **super**() ; en première ligne du constructeur de la sous-classe.
Donc attention il faut que dans la super-classe il existe un tel constructeur sans paramètre.
Les constructeurs sont chaînées et les appels aux constructeurs remontent de super-classe en super-classe jusqu'à la classe `Object`.
Dans l'ordre, le constructeur de la classe parent s'exécute avant le constructeur de la classe enfant.

si aucun constructeur de la sous-classe n'est défini, le compilateur ajoute le constructeur suivant :

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

Si un constructeur de la sous-classe oublie en première instruction l'appel à un constructeur de la super-classe, le compilateur ajoute l'instruction **super**() ; en première ligne du constructeur de la sous-classe.

Donc attention il faut que dans la super-classe il existe un tel constructeur sans paramètre.

Les constructeurs sont chaînées et les appels aux constructeurs remontent de super-classe en super-classe jusqu'à la classe `Object`. Dans l'ordre, le constructeur de la classe parent s'exécute avant le constructeur de la classe enfant.

si aucun constructeur de la sous-classe n'est défini, le compilateur ajoute le constructeur suivant :

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

Si un constructeur de la sous-classe oublie en première instruction l'appel à un constructeur de la super-classe, le compilateur ajoute l'instruction **super**() ; en première ligne du constructeur de la sous-classe.
Donc attention il faut que dans la super-classe il existe un tel constructeur sans paramètre.
Les constructeurs sont chaînées et les appels aux constructeurs remontent de super-classe en super-classe jusqu'à la classe `Object`.
Dans l'ordre, le constructeur de la classe parent s'exécute avant le constructeur de la classe enfant.

Polymorphisme

Tout objet a un type **déclaré** qui sera le type de sa référence.
Tout objet a un type **réel**, celui de son constructeur. Mais il peut appartenir à plusieurs classes.
Si A est la classe parent des classes B et C, alors une variable de type A peut être instanciée par un objet de type B ou C. On dit que le type A est *polymorphe*.
L'opérateur `instanceof` permet de tester l'appartenance d'un objet à une classe.

```
A a;// a est de type déclaré A
a = new B();// a est de type réel B
System.out.println("Objet de la classe A : " +
(a instanceof A));//true
System.out.println("Objet de la classe B : " +
 (a instanceof B));//true
System.out.println("Objet de la classe C : " +
(a instanceof C));//false
```

Conversion de classe

Si B hérite de A, un objet de type B est toujours un objet de type A mais un objet de type A n'est pas nécessairement un objet de type B.

```
A a;
B b;
a=b; // oui (upcasting implicite)
b=a; // ne compile pas
```

On peut convertir explicetement un objet B en objet A de la façon suivante :

```
a = (A) b;
```

Dans l'autre sens, si on est sûr que l'objet A peut être considéré comme un objet B, on peut faire une conversion de B vers A (downcasting).

On en verra l'utilité dans la méthode `clone()`.

Si une conversion est impossible une exception `CastClassException` est levée à l'exécution.

Redéfinir des méthodes

Règles :

- on ne peut pas redéfinir une méthode dans une sous-classe en modifiant le type de retour, en diminuant sa visibilité avec un modificateur de protection plus fort ou en la définissant de méthode d'instance à méthode statique ou inversement.
- une méthode redéfinie possède le même nom, les mêmes paramètres et le même type de retour

En cas de redéfinition d'une méthode, à l'exécution

- la méthode d'instance s'appliquera selon le **type réel** de l'objet qui l'appelle (*liaison tardive*)
- la méthode statique s'appliquera selon son **préfixe-classe**

exemple(suite)

```
// class Etudiant suite
public void afficheEtudiant(){// méthode redéfinie
System.out.println("nom : " + this.getNom()) ;
System.out.println("prenom : " + this.getPrenom()) ;
System.out.println("numero de carte d etudiant : "
+ this.getNumeroCarte()) ;
System.out.println("specialite : " + this.getSpecialite()) ;
System.out.println("annee : " + this.getAnneeDeFormation()) ;
}
```

Surcharge

Attention : ne pas confondre surcharge et redéfinition.
Une méthode est surchargée si elle a le même nom mais des paramètres différents en type, en nombre ou en ordre.
L'appel a une méthode d'instance surchargée se fait selon le type déclaré de l'objet appelant.

Masquage d'attributs

Une sous-classe peut définir un attribut de même nom que sa classe parent : il y a masquage d'attribut.
Une variable d'instance masquée utilisée dans une classe est la variable d'instance connue statiquement selon la **classe** de l'objet.
A EVITER

Object

Toute classe hérite de la classe `Object`.

- Constructeur : `Object();`
- Méthodes :
  - `public String toString();`
  - `public boolean equals (Object obj);`
  - `protected Object clone();`
  - ...

toString

La méthode d'instance `toString` renvoie le nom de la classe de l'objet suivi de la référence de cet objet. Cette méthode est utilisée par la méthode `System.out.print`.
On peut la redéfinir.

toString

```
public class Point{
float abs, ord;
Point(float x, float y){ this.abs =x;this.ord = y;}
public String toString(){
return("abscisse:"+this.abs+",ordonnée:"+this.ord);}}
...
Point monPoint = new(2,1);
System.out.println(monPoint) ;
//abscisse : 2.0, ordonnée :1.0
```

Une sous-classe peut redéfinir la méthode `toString` en utilisant la redéfinition de sa superclasse en appelant `super.toString`.

equals

== compare

- les valeurs pour les types primitifs
- les références pour les objets ou les tableaux

Si on veut une comparaison en « profondeur » , il est nécessaire de redéfinir ==.

```
public boolean equals(Point autrePoint){
return(this.abs==autrePoint.abs && this.ord==autrePoint.ord);}
```

equals

== compare

- les valeurs pour les types primitifs
- les références pour les objets ou les tableaux

Si on veut une comparaison en « profondeur » , il est nécessaire de redéfinir ==.

```
public boolean equals(Point autrePoint){
return(this.abs==autrePoint.abs && this.ord==autrePoint.ord);}
```