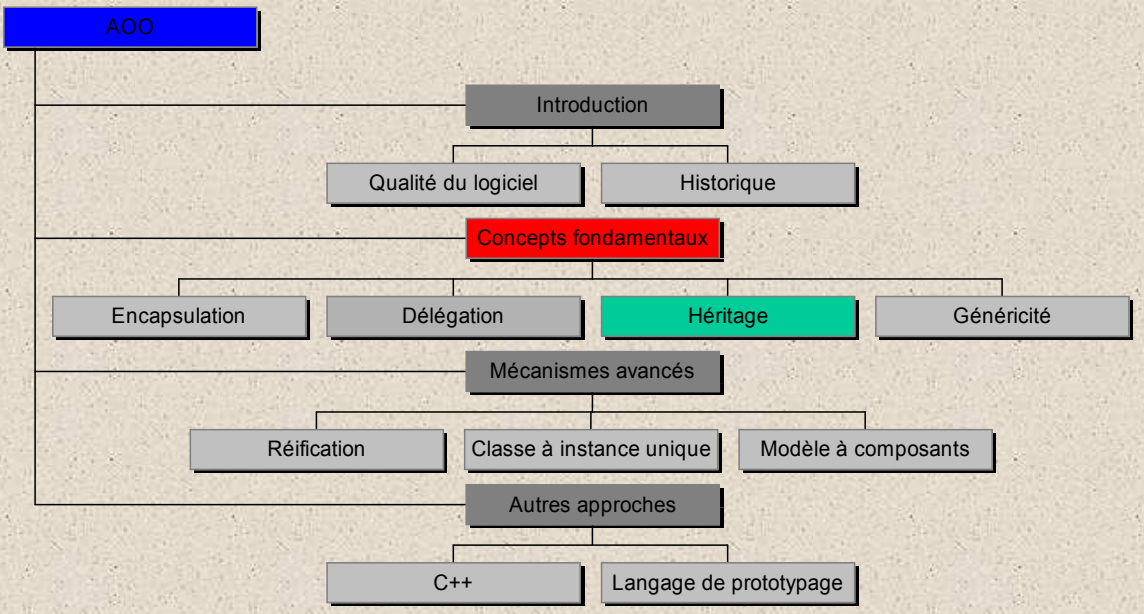
L'HERITAGE

L'HERITAGE MULTIPLE

Sommaire



Sommaire

Héritage multiple & Java

Ce chapitre a peu d'intérêt pour Java, cependant IL me paraît essentiel dans le cadre de la Conception Orientée Objet, là est la raison de sa présence dans ce cours.

Vive les classes miscellanées

L'héritage multiple

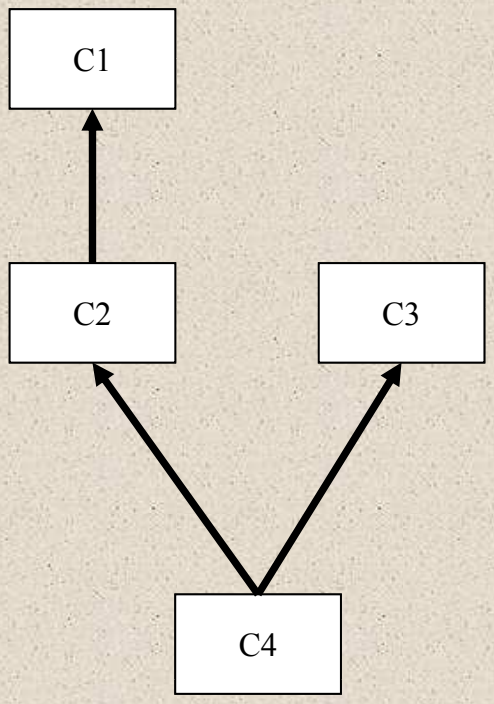
L'héritage est dit multiple lorsqu'une classe peut hériter de plusieurs autres classes.

Cette possibilité prend tout son sens lorsque l'on admet que l'héritage est une relation qui offre plusieurs sémantiques de liaison inter-classes.

- Chaque classe décrit une abstraction, une classe peut être l'union de ces différentes abstractions.

Problèmes induits

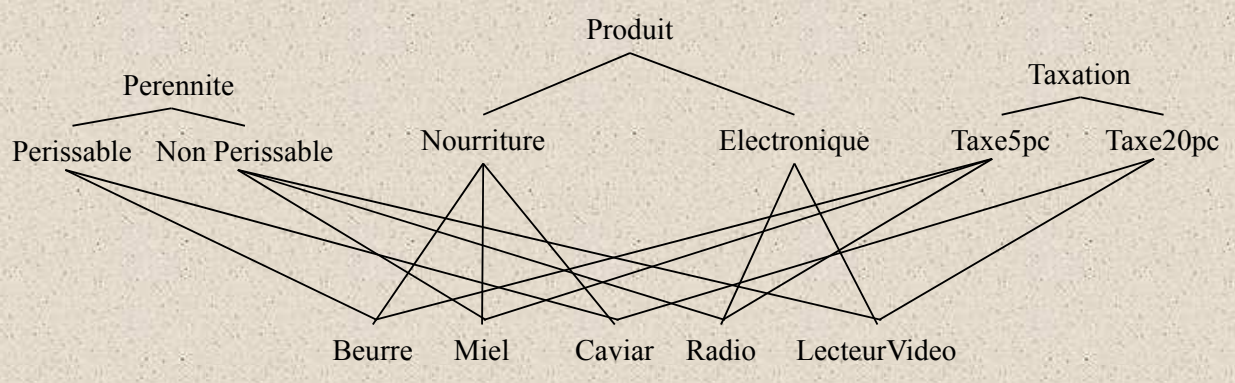
- Synonymie
- Introduction de circuits



La pluralité naturelle

L'absence d'héritage multiple ou de moyen équivalent est une restriction excessivement pénalisante

- Dans le monde réel les éléments complexes tirent leurs propriétés de plusieurs origines.
- Il est difficile d'imaginer que l'on peut tout ranger dans une seule classification



Il existe une autre technique pour assurer l'aspect multi-facettes des choses.

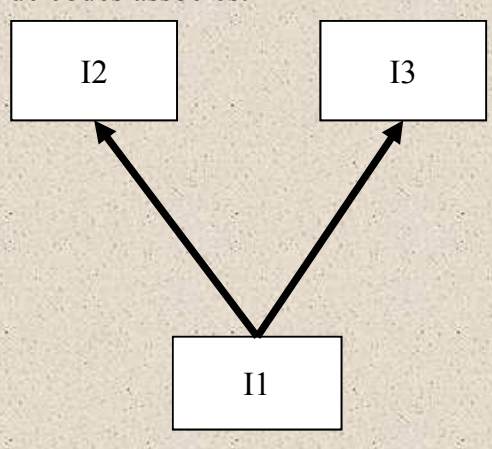
- La multi-instanciation qui permet de créer un objet instance de plusieurs classes.
  - héritage multiple dynamique qui est souvent utilisé dans les langages non typés.

Héritage multiple en Java

Seul le concept d'interface supporte le principe de l'héritage multiple

– Cette restriction est excessivement simplificatrice pour :

- la compilation, car le concept d'interface se réduit à une liste de profils de méthodes abstraites et ne nécessite pas de gestion de codes associés.



- la liaison dynamique continue à s'effectuer sur la structure arborescente des classes

Problème de la synonymie

```
void plus(int v) {
//Ensures : Binf=_Binf+v
//          Bsup=_Bsup+v
…}
```

```
void plus(Entier v) {
//Ensures : Valeur = _Valeur+V.Valeur
…}
```

```
void plus(Intervalle v) {
//Ensures : Binf=min(_Binf+v.Binf)
//          Bsup=max(_Bsup+v.Bsup)
…}
```

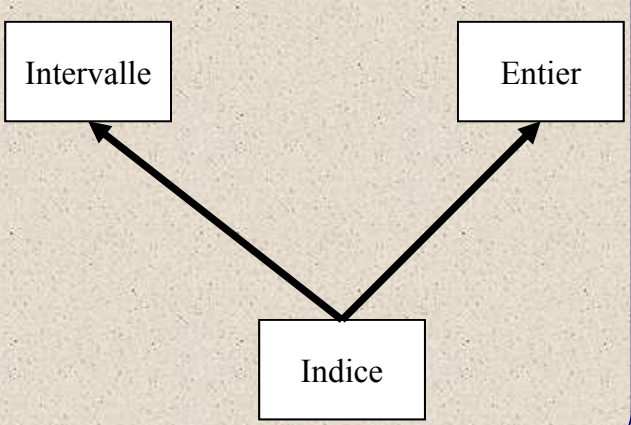
```
void plus(int v) {
//Ensures : Valeur = _Valeur+v

…}
```

Les différents héritages fournissent des compétences distinctes qui peuvent être dénotées par le même nom

Solutions

– Surcharge
– Ordonnancement
– Renommage

Mécanisme de fusion



Surcharge

```
void plus(int v) {
//Ensures : Binf=_Binf+v
//          Bsup=_Bsup+v
…}
```

Une incompatibilité demeure

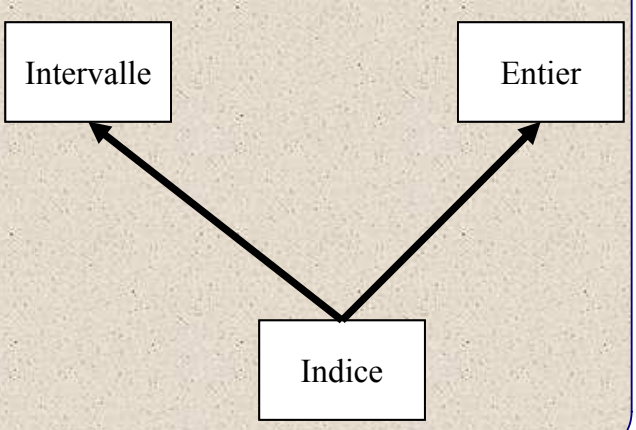
```
void plus(Entier v) {
//Ensures : Valeur = _Valeur+V.Valeur
…}
```

```
void plus(Intervalle v) {
//Ensures : Binf=min(_Binf+v.Binf)
//          Bsup=max(_Bsup+v.Bsup)
…}
```

```
void plus(int v) {
//Ensures : Valeur = _Valeur+v

…}
```

Les méthodes sont sélectionnées en fonction de leur nom mais aussi en fonction du nombre et du type de leurs paramètres.

- Le problème de synonymie est donc restreint, mais pas nécessairement supprimé.
- Ce mécanisme est utile pour décrire des méthodes dont la sémantique est la même, mais dont les paramètres d'entrée diffèrent.



Ordonnancement des héritages

```
void plus(int v) {
//Ensures : Binf=_Binf+v
//          Bsup=_Bsup+v
…}
```

```
void plus(Entier v) {
//Ensures : Valeur = _Valeur+V.Valeur
…}
```

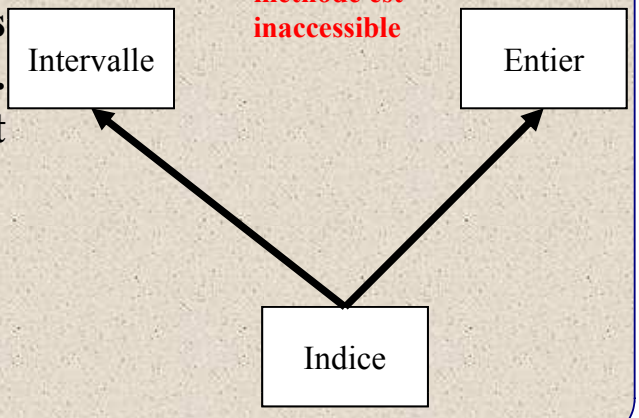
```
void plus(Intervalle v) {
//Ensures : Binf=min(_Binf+v.Binf)
//          Bsup=max(_Bsup+v.Bsup)
…}
```

```
void plus(int v) {
//Ensures : Valeur = _Valeur+v

…}
```

Cette méthode est inaccessible

L'ordre d'énumération des héritages établit une priorité. La première méthode est choisie.

- linéarisation



Renommage

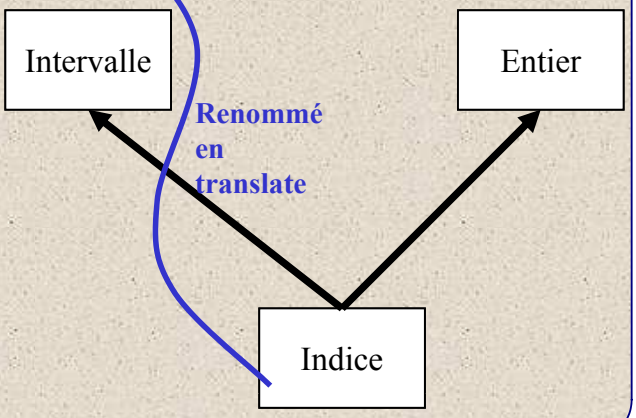
```
void plus(int v) {
//Ensures : Binf=_Binf+v
//          Bsup=_Bsup+v
…}
```

La surcharge n'est pas autorisée

```
void plus(int v) {
//Ensures : Valeur = _Valeur+v

…}
```

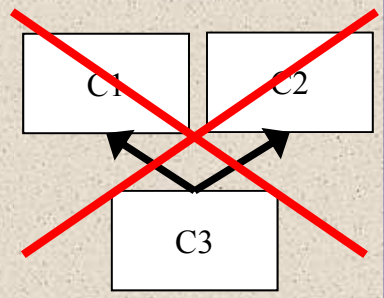
On laisse au concepteur le soin de régler lui-même les conflits de nom à l'aide d'un mécanisme de renommage.

- Ce mécanisme porte sur la relation d'héritage qui lie une classe descendante à une classe ancêtre. Ce qui laisse la classe mère inchangée.
- Il empêche la surcharge
- Il peut être aussi le support de l'affinement d'un nom quand il en existe un plus pertinent.



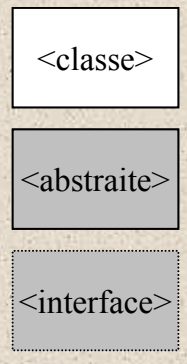
Héritage multiple & Java

L'héritage de multiple classes n'est pas autorisé en Java qui n'accepte que l'héritage simple



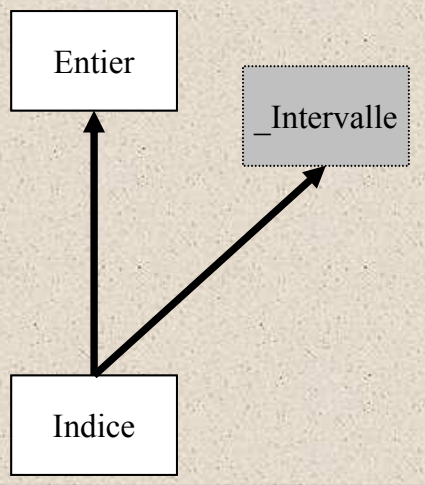
Réalisation d'un héritage multiple à l'aide de :

- classe
- interface
- Ne fournit pas toute la puissance de l'héritage multiple de classes.
- Oblige à des contorsions

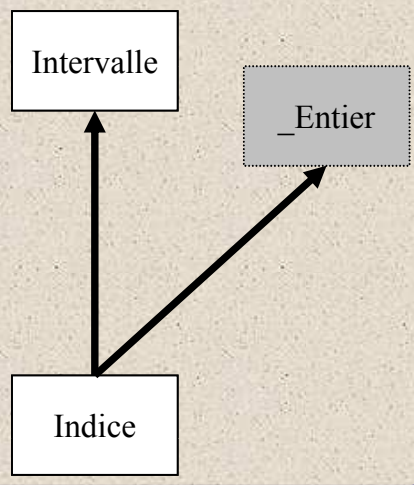


Solutions dissymétriques

On privilégie la relation à Entier en considérant qu'un Indice correspond plus à une spécialisation de Entier



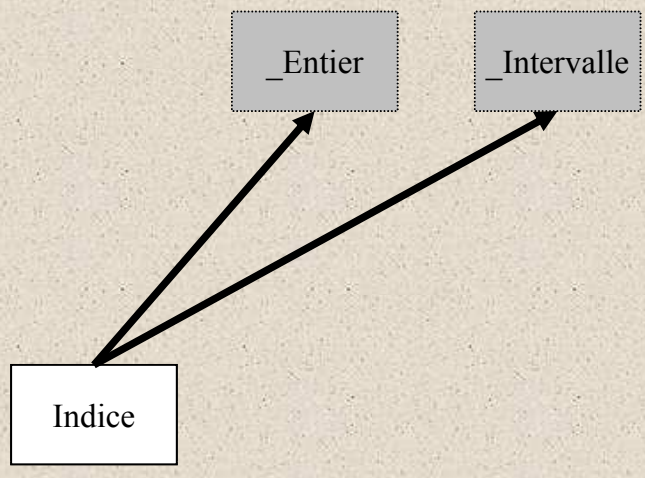
On privilégie la relation à Intervalle en considérant qu'un Indice correspond plus à une spécialisation de Intervalle



Solution symétrique

On ne privilégie aucune des 2 relations

– le système n'est pas récurrent
- une interface ne peut hériter d'une classe
- on passe du monde des classes dans celui des interfaces.

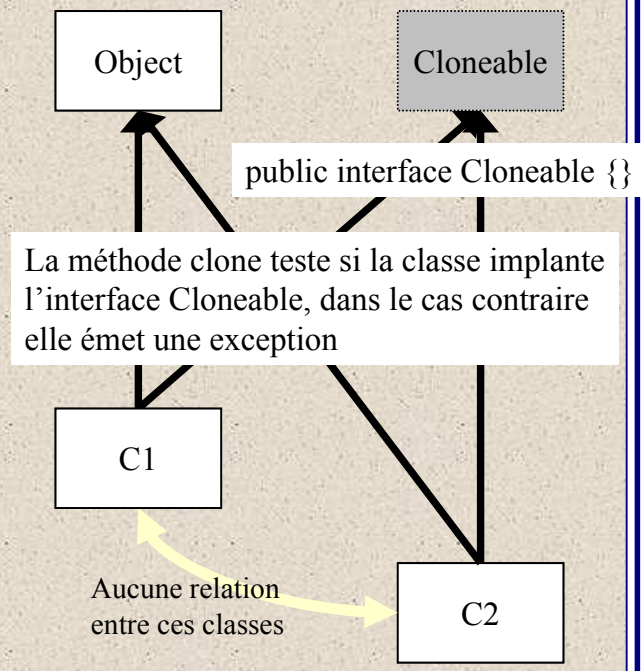


Héritage de comportement

On utilise une interface pour modéliser une propriété

On utilise l'«héritage » pour associer la propriété à la classe.

- Toute classe ayant la propriété hérite de l'interface définissant cette propriété (comportement).
- L'interface Cloneable donne à la classe qui l'implante la propriété de pouvoir dupliquer ses instances par l'utilisation de la méthode clone() dont une définition se trouve dans Object.
- Cloneable définit une poignée sur des objets que l'on pourra de facto cloner

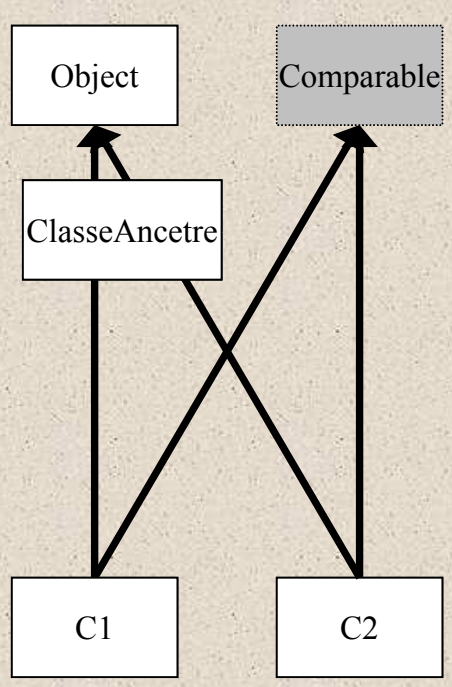


- Cependant en Java, la vacuité des interfaces pose qqc problèmes.

Comparable

```
public interface Comparable {
  /**
  * définit la propriété de l 'existence d'un
  * ordre sur les instances de la classes dérivée
  */
    – public boolean egal(Object other);
    – public boolean superieur(Object other);
    – public boolean superieurEgal(Object other);
    – public boolean inferieur(Object other);
    – public boolean inferieurEgal(Object other);
    – public boolean different(Object other);
}
```

- Le problème provient de l'impossibilité de décrire, dans l'interface, les méthodes autres que «egal» et «superieur»
- On ne peut pas utiliser une classe abstraite à cause de l 'héritage simple des classes.

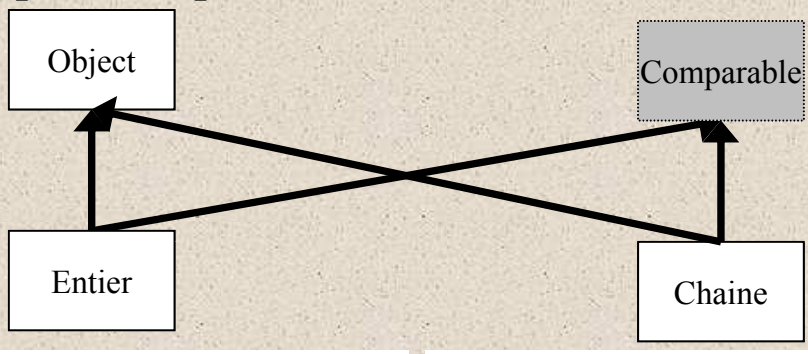


Entier

```
class ENTIER implements Comparabale {
  public ENTIER(int Valeur) {Set(Valeur);}
  …
  public boolean egal(Object other) {
    return Valeur==other.Valeur();}
  public boolean superieur(Object other) {
    return Valeur>other.Valeur();}
  public boolean superieurEgal(Object other) {
    return susperieur(other) || egal(other);}
  public boolean inferieur(Object other) {
    return other.susperieur(this);}
  public boolean inferieurEgal(Object other) {
    return other.susperieurEgal(this);}
  public boolean different(Object other) {
    return !egal(other);}

  private int Valeur;
}
```


Interface & Evolution

L'utilisation d'interface impose une duplication du code, puisque celui-ci ne peut être placé au bon niveau d'abstraction.



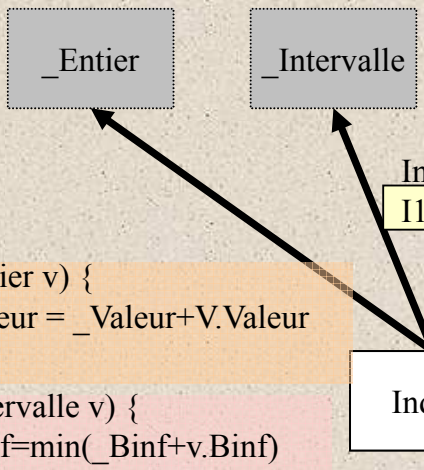

```
...
public boolean superieurEgal(Object other) {
        return susperieur(other) || egal(other);}
public boolean inferieur(Object other) {
        return other.susperieur(this);}
public boolean inferieurEgal(Object other) {
        return other.susperieurEgal(this);}
public boolean different(Object other) {
        return !egal(other);}
...
```

```
...
public boolean superieurEgal(Object other) {
        return susperieur(other) || egal(other);}
public boolean inferieur(Object other) {
        return other.susperieur(this);}
public boolean inferieurEgal(Object other) {
        return other.susperieurEgal(this);}
public boolean different(Object other) {
        return !egal(other);}
...
```


Héritage multiple & Surcharge

Le concept d'interface offre un succédanée d'héritage multiple.

- Cette forme, bien que restreinte, n'en comporte pas moins des problèmes
- La combinaison avec la surcharge peut être à l'origine d'incohérence comme précédemment vu




```
Indice I1= new Indice(…), I2 = new Indice(…);
I1.plus(I2);
```

```
void plus(_Entier v) {
//Ensures : Valeur = _Valeur+V.Valeur
…}
```

```
void plus(_Intervalle v) {
//Ensures : Binf=min(_Binf+v.Binf)
//          Bsup=max(_Bsup+v.Bsup)
…}
```

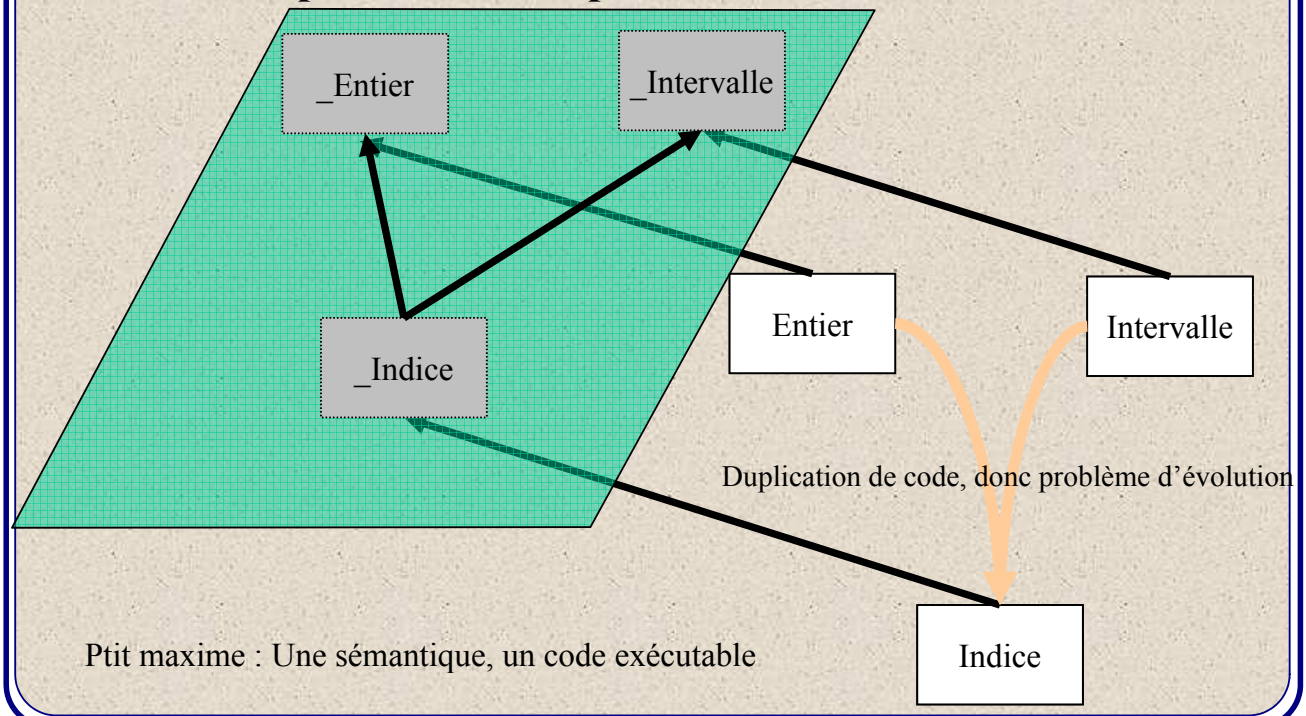
```
I1.plus((_Intervalle)I2);
```

≠

```
I1.plus((_Entier)I2);
```


Héritage multiple & interface

Une interface peut hériter de plusieurs autres interfaces



Duplication de code, donc problème d'évolution

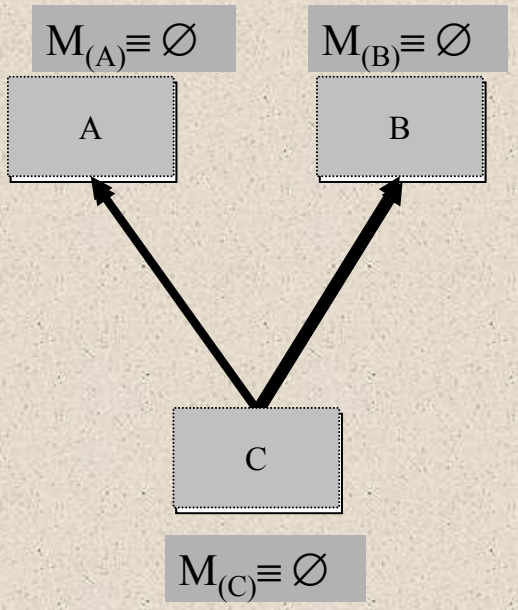
Ptit maxime : Une sémantique, un code exécutable

Fusion

Cas particulier de synonymie de méthodes ayant le même profil.

Deux méthodes $M_{(A)}$ et $M_{(B)}$ peuvent fusionner en $M_{(C)}$ si :

- C hérite de A et B
- Les profils de $M_{(A)}$ et $M_{(B)}$ sont unifiables
  - les types des arguments sont mutuellement comptatibles
- Les sémantiques de $M_{(A)}$ et $M_{(B)}$ sont unifiables
  - L'une ou l'autre des méthodes est abstraite ($M^A = \varnothing \vee M^B = \varnothing$)
- Si ces conditions sont remplies alors :
  - $M^C = M^A \cup M^B$
- $M_{(A)} \equiv M^A \wedge M_{(B)} \equiv \varnothing$
- $M_{(A)} \equiv \varnothing \wedge M_{(B)} \equiv M^B$
- $M_{(A)} \equiv \varnothing \wedge M_{(B)} \equiv \varnothing$



En java seules les interfaces peuvent multi-hériter, une classe peut hériter d'une interface et d'une classe

"Indéfinition" & Fusion

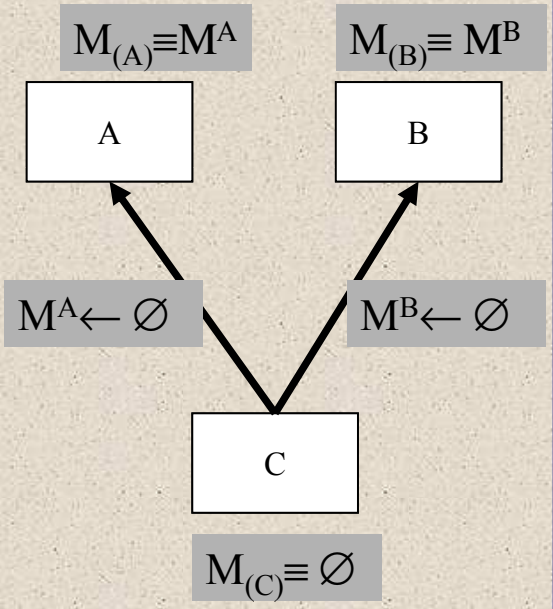
Si les codes de $M_{(A)}$ et $M_{(B)}$ ne sont pas unifiables

- $M^A \neq \varnothing \wedge M^B \neq \varnothing$

Mais les profils sont unifiables

L'"indéfinition" permet de forcer la fusion

- Indéfinition de $M_{(A)}$
- Indéfinition de $M_{(B)}$
- Indéfinition de $M_{(A)}$ et $M_{(B)}$

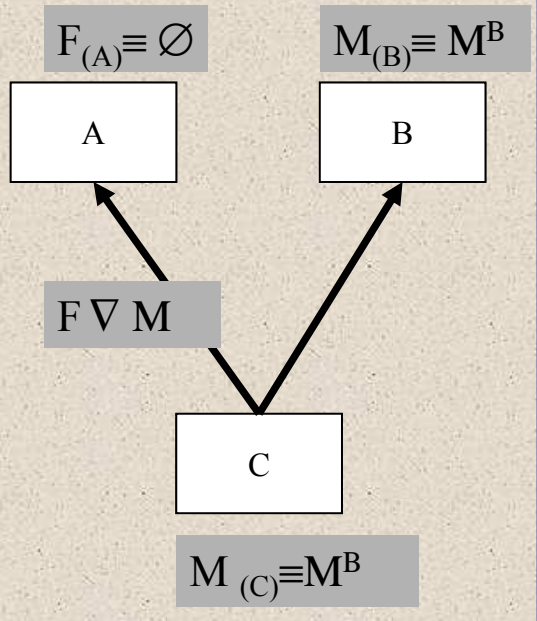


Cet opérateur n'existe pas en Java, puisque les interfaces sont toujours abstraites

Renommage & Fusion

Deux méthodes $F_{(A)}$ et $M_{(B)}$ peuvent fusionner en $M_{(C)}$ ou $F_{(C)}$ ou $K_{(C)}$ si :

- C hérite de A et B
- Les profils de $F_{(A)}$ et $M_{(B)}$ sont unifiables
- Les sémantiques de $F_{(A)}$ et $M_{(B)}$ sont unifiables ou sont forcées à l'être
- On renomme l'une ou l'autre ou les deux méthodes comme il se doit
  - $M^C = F^A \cup M^B$
- A refA = new C(…);
  refA.F(…) ↙ M^B



Fusion : synthèse

	$M_B = \varnothing$	$M_B = M^B$		
$M_A = \varnothing$	$M_C = \varnothing$ {1,2,2'}	$M_C = M^B$ {2',3'}		
$M_A = M^A$	$M_C = M^A$ {2,3}	redéfinition	$M_B \leftarrow M^B$	$M_B \leftarrow \varnothing$
		$M_A \leftarrow M^A$	non autorisé	$M_C = M^A$
		$M_A \leftarrow \varnothing$	$M_C = M^B$	$M_C = \varnothing$

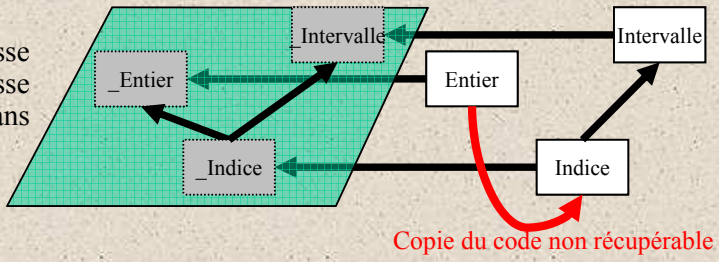
Interface(C)	Interface(A)	Abstraite(A)	Concrète(A)
Interface(B)	1	Non autorisé	
Abstraite(B)	Non autorisé		
Concrète(B)			

Classe(C)	Interface(A)	Abstraite(A)	Concrète(A)
Interface(B)	1	2	3
Abstraite(B)	2'	Non autorisé	
Concrète(B)	3'		

Héritage multiple & implantation Java

Implantation d'un héritage multiple par duplication.

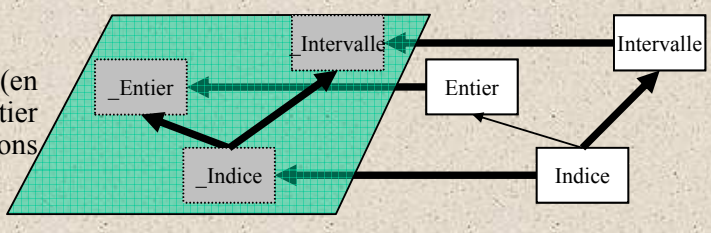
– On duplique dans le code de la classe Indice le code réalisant dans la classe Entier les abstractions définies dans l'interface _Entier.



Remarque : Aucune cohérence n'est assurée entre les codes des classes Entier et Indice

Implantation d'un héritage multiple par délégation.

– Une instance de la classe Indice utilise (en interne) une instance de la classe Entier pour assurer la réalisation des abstractions définies dans l'interface _Entier.



Remarque : Oblige la classe Entier à fournir les services nécessaires à la classe Indice, on ajoute un appel de méthode supplémentaire.

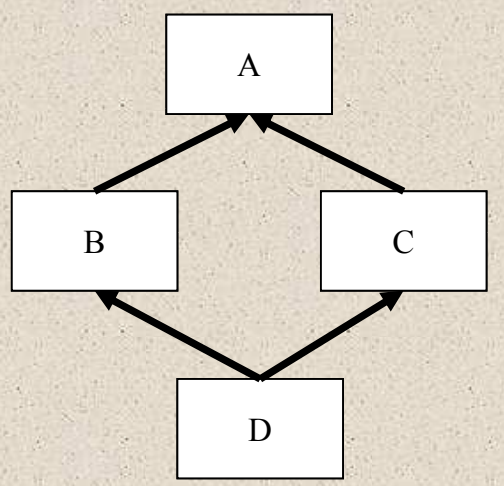
Introduction de circuits

La relation n'est plus strictement arborescente, mais devient un DAG. Une classe peut hériter plus d'une fois d'une autre classe.

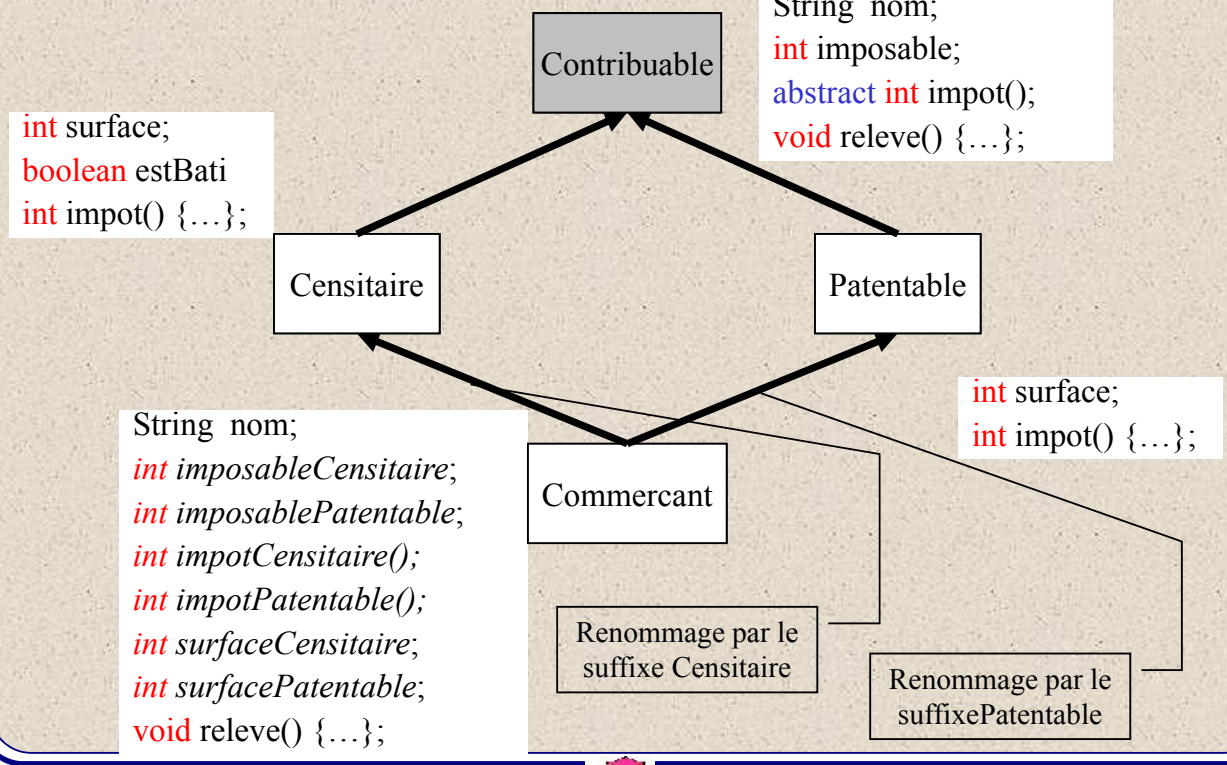
Une classe par le biais de l'héritage multiple hérite d'une même classe par des chemins différents.

– comment interpréter les conflits de noms qui en découlent ?

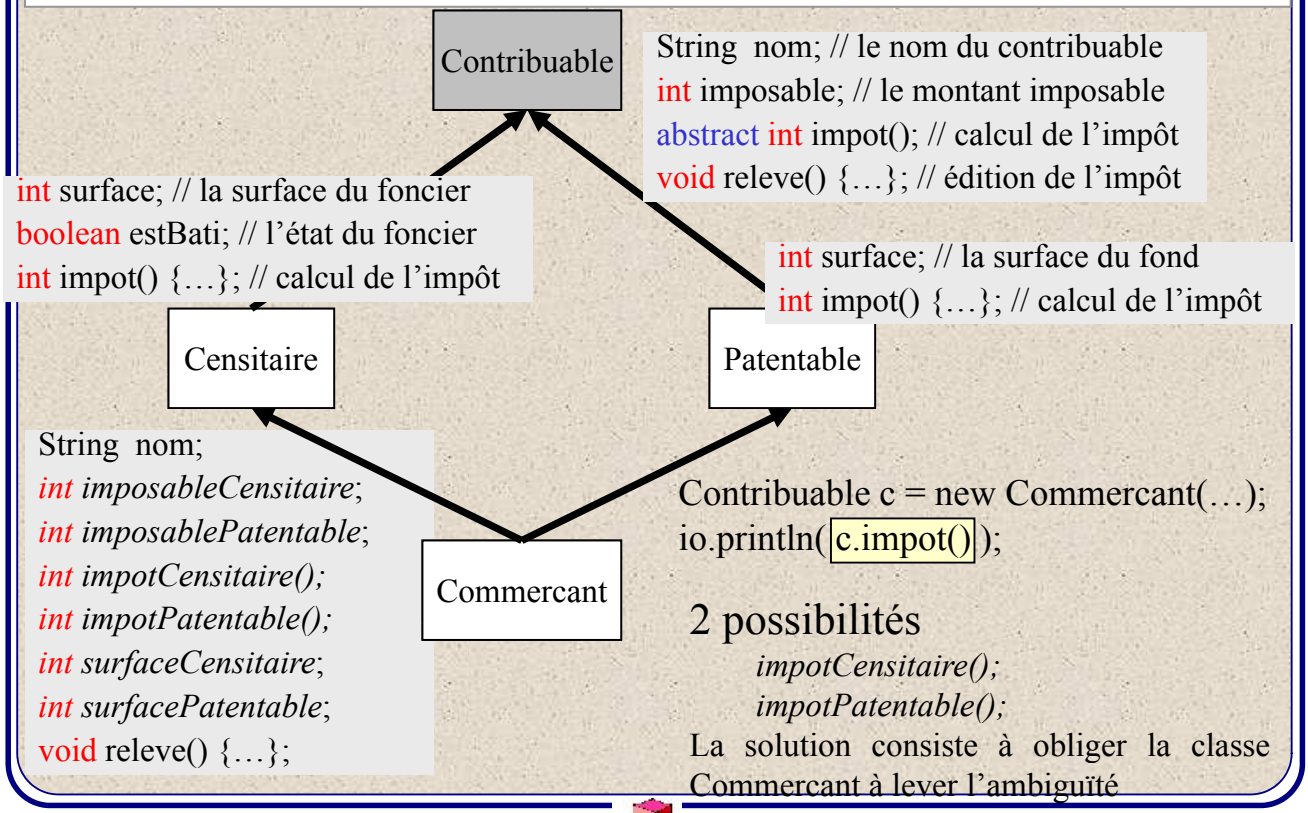
- non duplication (partage)
  - identité
  - nécessite une compatibilité réciproque
- duplication
  - renommage



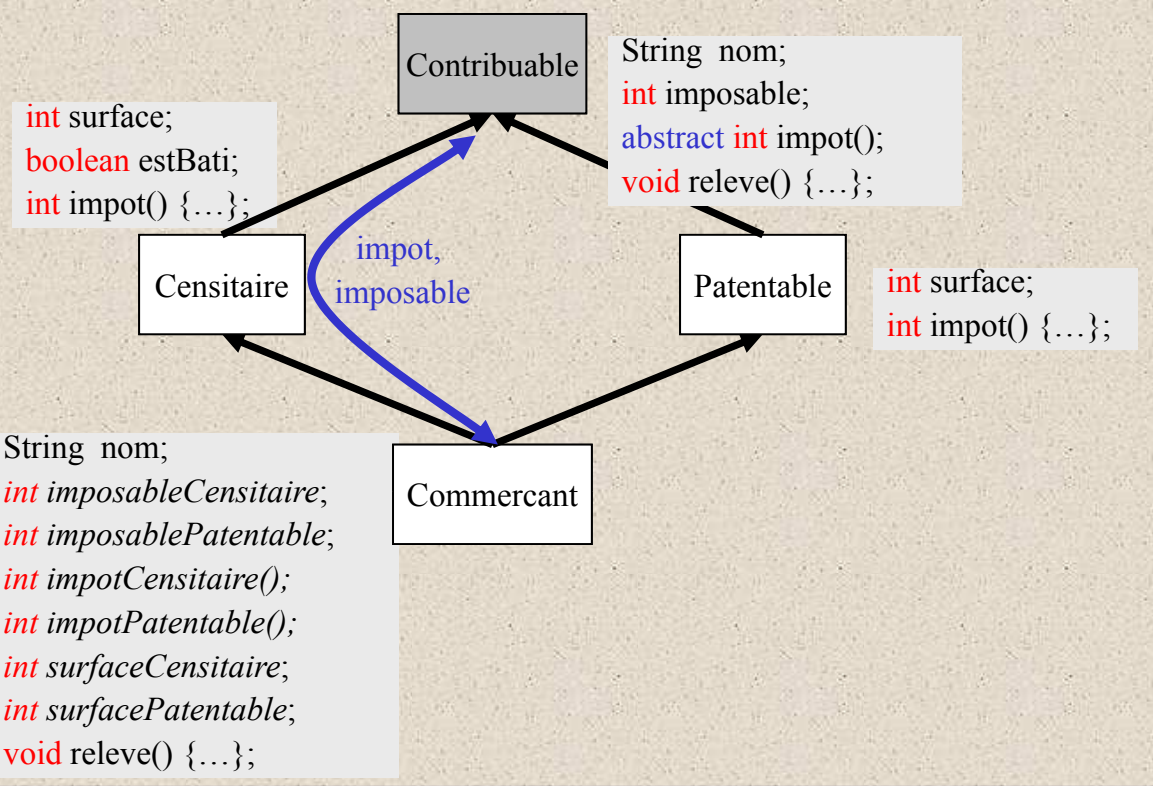
Héritage répété indirect



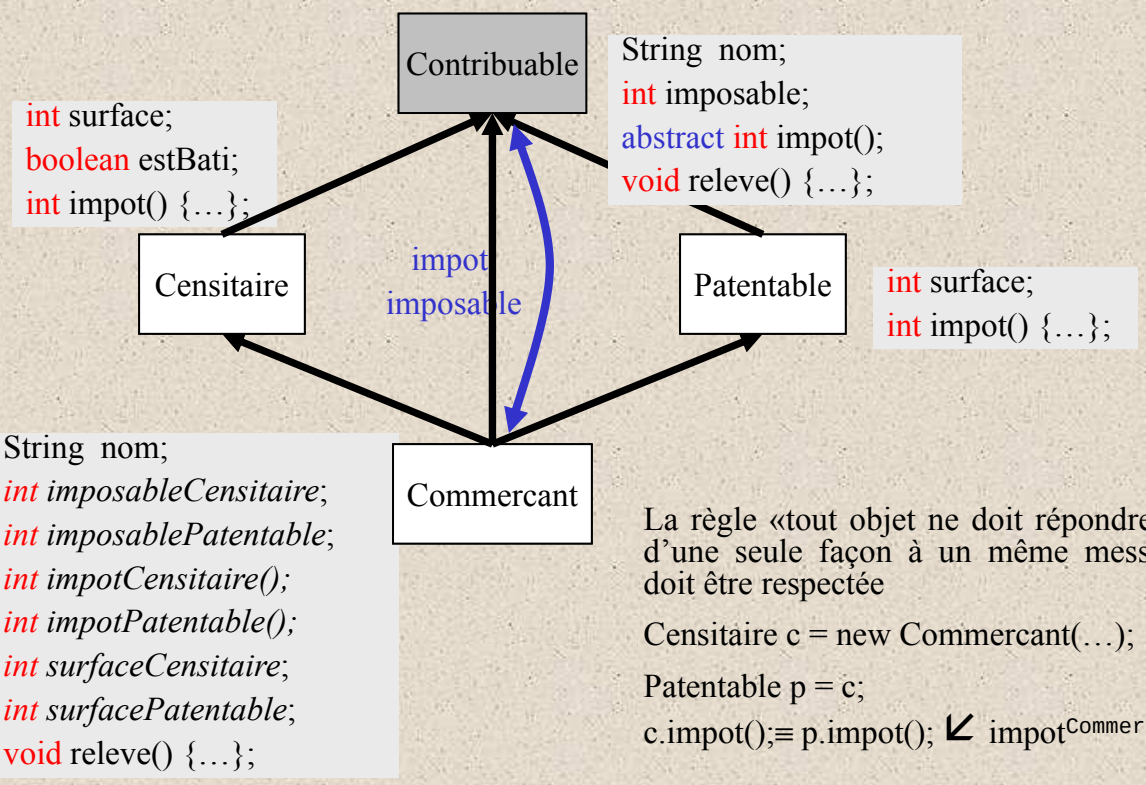
Héritage répété : problème



Héritage répété : choix d'un chemin



Héritage répété : nouvelle définition



La règle «tout objet ne doit répondre que d'une seule façon à un même message» doit être respectée

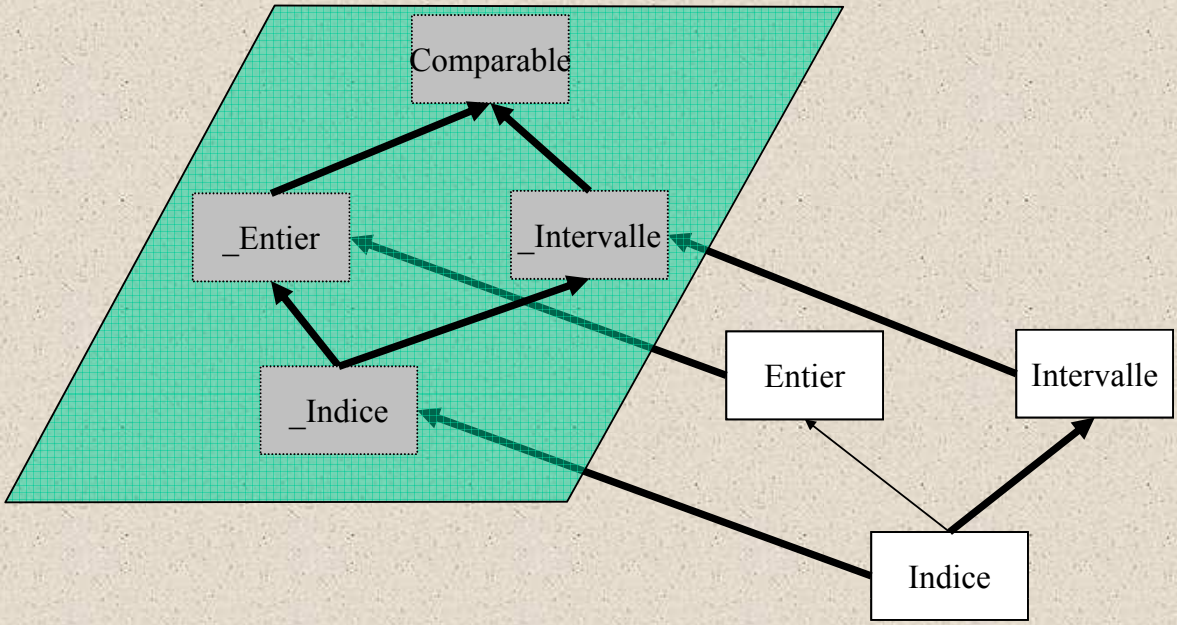
Censitaire c = new Commercant(…);

Patentable p = c;

c.impot();≡ p.impot(); ↙ impot`Commercant`

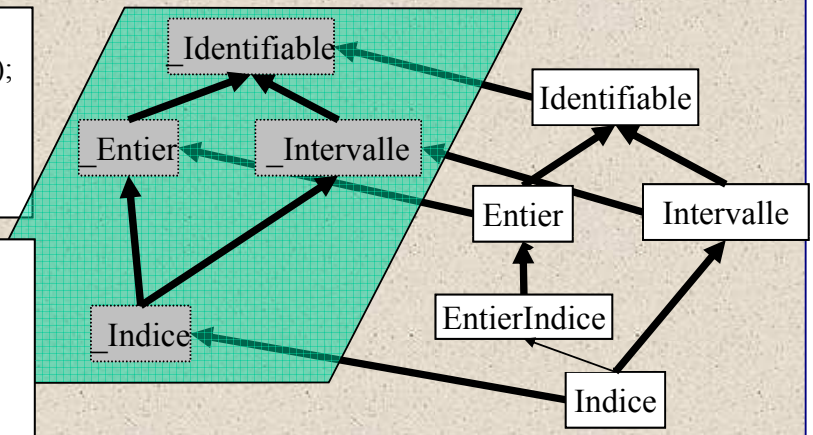
Héritage répété & Java

L'héritage répété est possible en Java sur les interfaces, mais sans réel intérêt. Il relève du cas de non duplication.



Héritage répété & Java

```
interface _Identifiable {
   public void identite (String identite);
   public String identite() ;
   …
}
```




```
class Identifiable {
   private String identite;
   public Identifiable (){identite("");}
   public void identite (String identite){
      this.identite=identite;
   }
   public String identite() {return identite;}
   ….      // code intervenant sur identite via
           // ses méthodes de manipulation.
}
```

```
class EntierIndice extends Entier {
   private Indice indice;
   public EntierIndice(Indice indice){ super(); this.indice=indice;}
   public void identite (String identite){indice.identite(identite);}
   public String identite() {return indice.identite();}
}
```

Remarque : Cette solution n'empêche pas la duplication des structures de données,

Remarque : Le système échoue à l'exécution car on crée un cycle dans la création des instances.