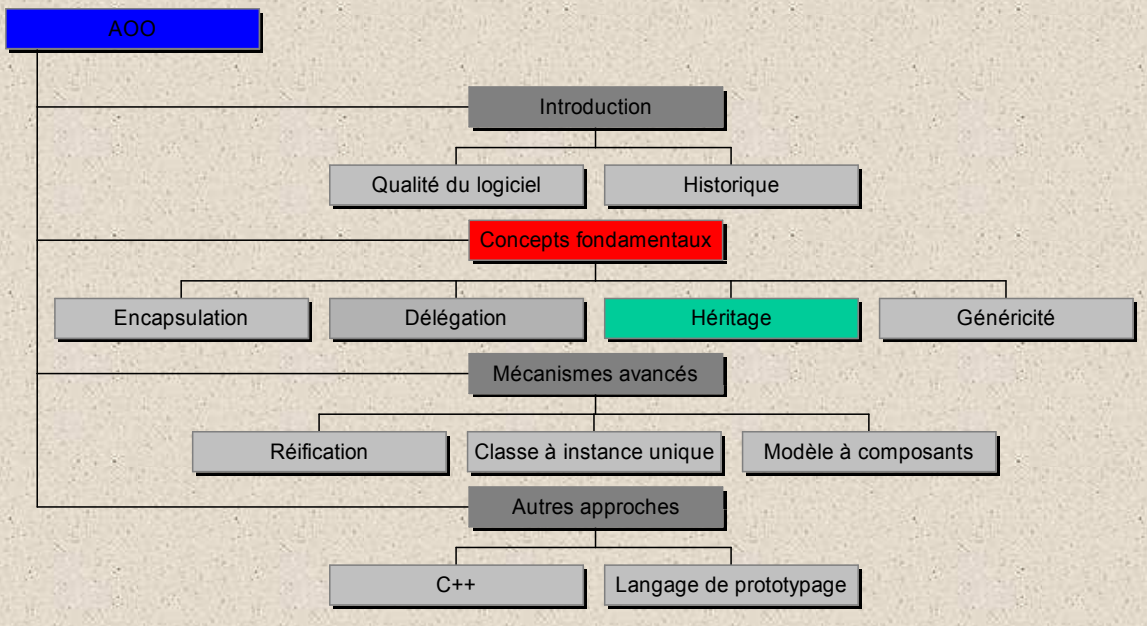
L'HERITAGE

Héritage simple

Sommaire

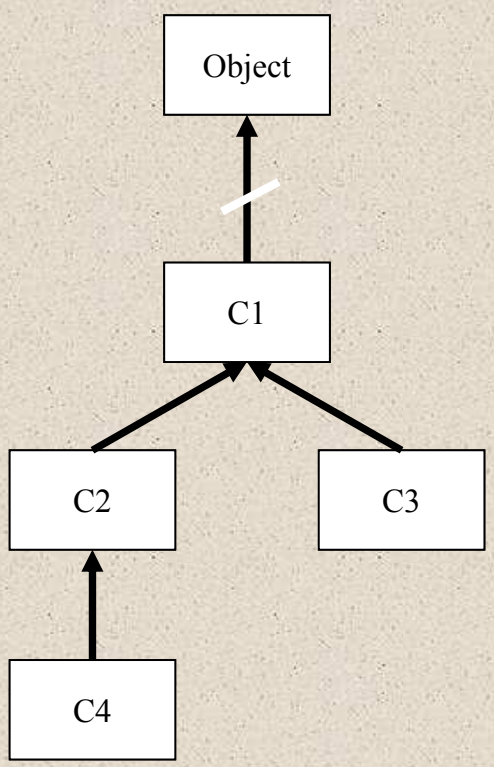


Sommaire

L'héritage simple

L'HERITAGE

- Moyen de description incrémentale des caractéristiques d'une classe d'objets par rapport à une autre classe.
- Tout objet de la classe Fille **se comporte** au moins comme ceux de la classe Mère
- Cette relation est transitive
- Toute classe hérite implicitement d'une classe racine en l'absence d'un autre héritage
- Les maîtres mots
  - réutilisation
  - polymorphisme (flexibilité)



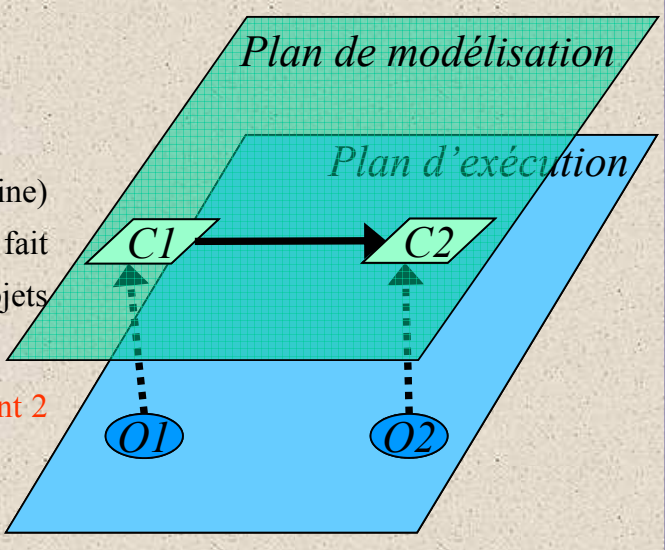
Patrimoine transmis

Soient C1 et C2 deux classes

- O1 est instance de C1
- O2 est instance de C2

C1 hérite de C2

- l'ensemble des caractéristiques (patrimoine) qui définit un objet instance de C2 fait partie de l'ensemble définissant les objets instances de C1
- Il n'y a pas de relation particulière reliant 2 instances de classes liées par l'héritage



Les interprétations de l'héritage

La classe A hérite de la classe B

point de vue ensembliste (types)

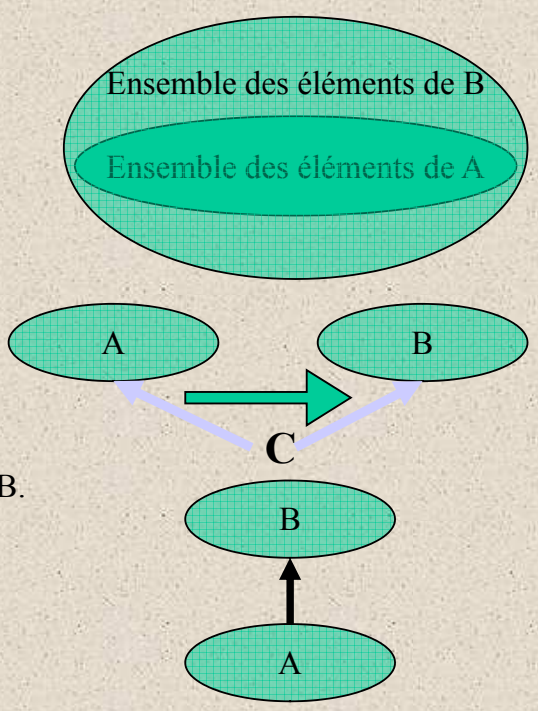
- $\forall C, C \in A \Rightarrow C \in B \ (A \subset B)$
- Les éléments de A sont aussi des éléments de B
- Correspond au principe de vérification des types

point de vue logique (prédicat)

- $\forall C, A(C) \Rightarrow B(C)$
- Si C est un A alors C est aussi un B
- C est un objet polymorphe

point de vue conceptuel

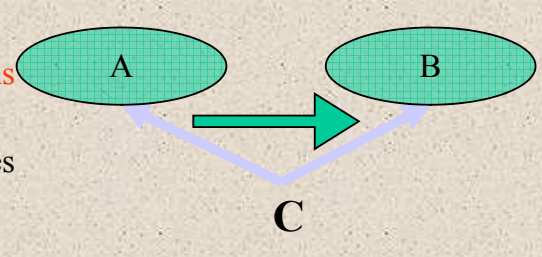
- Le concept A est plus spécifique que le concept B.
- A est une sorte de B.
- L'héritage fait une classification



Les différentes vues de l'héritage

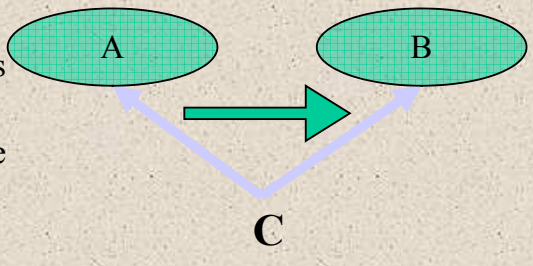
point de vue comportemental

- Les méthodes applicables aux instances de B le sont à celles de A.
- Les instances de A se comportent au moins comme celles de B.
- L'arbre d'héritage constitue une classification des capacités.



point de vue structurel

- Les attributs de B font partie des attributs de A.
- Les instances de A possèdent au moins les attributs de celles de B.
- n'est qu'est cas particulier du point de vue précédent



ENTIER

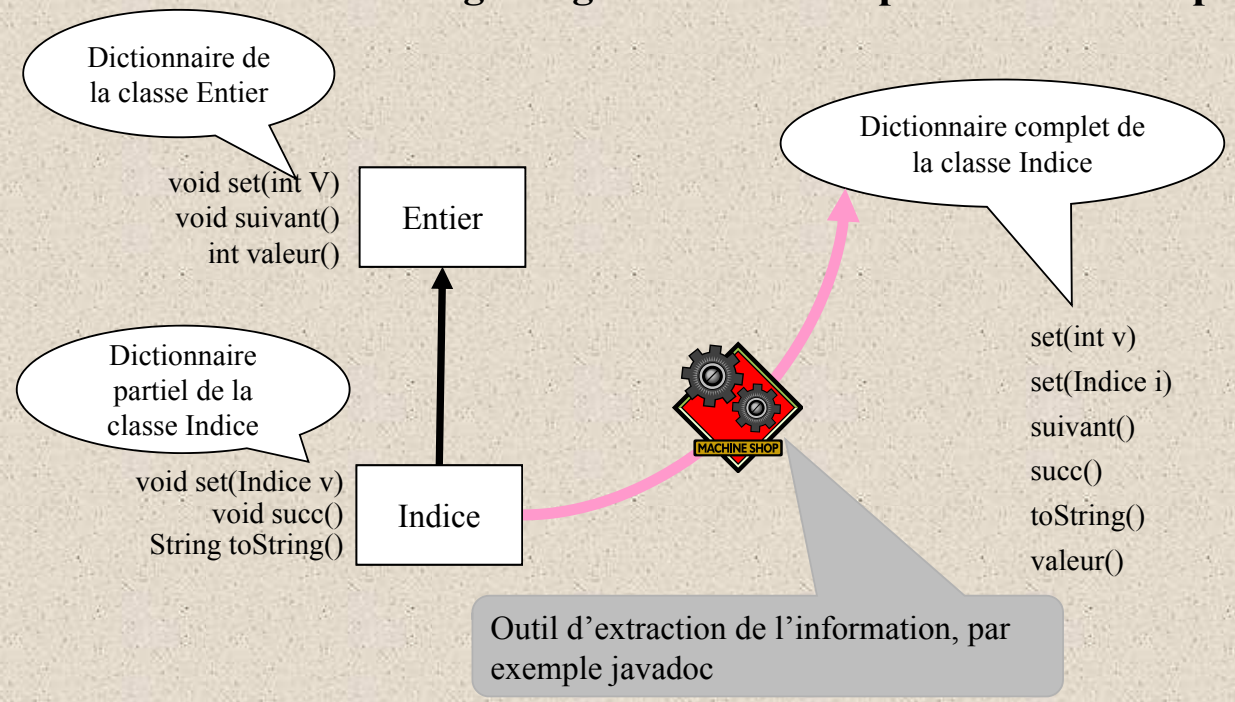
```
class Entier {
    public Entier(int valeur) {set(valeur);
    }
    /**
    * @fonction: affecte l'entier avec la valeur du paramètre
    */
    public void set(int valeur) {this.valeur = valeur;
    }
    /**
    * @fonction: fait progresser l'entier d  'une unité
    */
    public void suivant() { valeur++;
    }
    /**
    * @fonction: restitue la valeur courante de l'entier
    */
    public int valeur() {return valeur;
    }
    private int valeur;
}
```

INDICE

```
public class Iindice extends Entier {
    /**
    * @fonction: création d'un indice dans l'intervalle [binf..bsup]
    */
     public Indice(int binf,bsup) {this.binf=binf; this.bsup=bsup;super.set(binf);
    }
    /**
    * @fonction: affecte l'indice avec la valeur de l'indice argument. celle-ci doit
    *    respecter les bornes
    */
    public void set(Indice i) {set (i.valeur());
    }
    /**
    * @fonction: fait progresser l'indice d'une unité s'il n'est pas sur bsup
    */
     public void succ() {if(valeur  < bsup) valeur++;
    }
    /**
    * @fonction: retitue la représentation textuel de l'objet
    */
    public String  toString() {
        return '['+binf+'/'+valeur+'/'+bsup+']';
    }
    // Valeur et bornes de l'indice.
     private int binf, bsup;
}
```


Modèle complet d'une classe

L'utilisation de l'héritage fragmente la description d'un concept



Compatibilité de types

Utilité des types

- Permettre de vérifier le plus de propriétés possibles afin d'assurer que l'exécution du programme ne produira pas de comportements anormaux.
- Fournir un cadre permettant de mieux formaliser les informations manipulées.

Servitude des types

- Oblige le concepteur à faire un travail de formalisation.
- Oblige à des contorsions lorsqu'ils ne permettent pas de faire ce que l'on souhaite.

Types et héritage

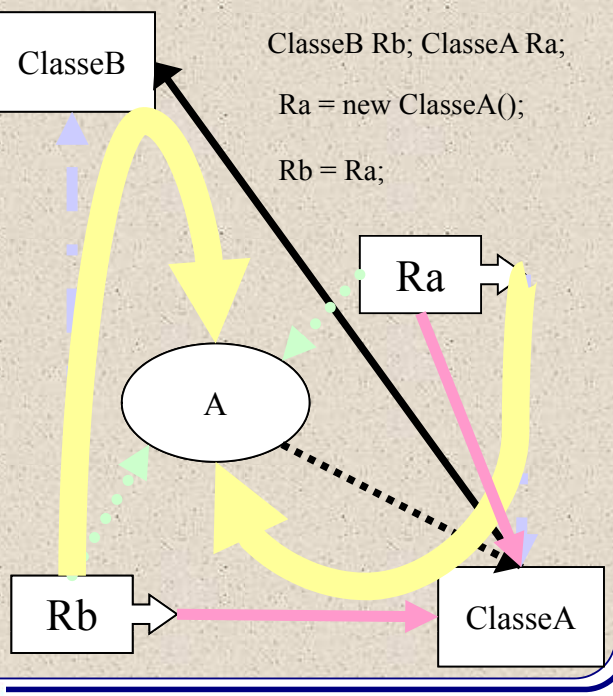
- La compatibilité de types est supportée par la relation d'héritage
- Le polymorphisme
  - Typage non coercitif
  - Ne concerne que les références

Compatibilité de types et héritage

Tout objet est manipulé au travers de deux types

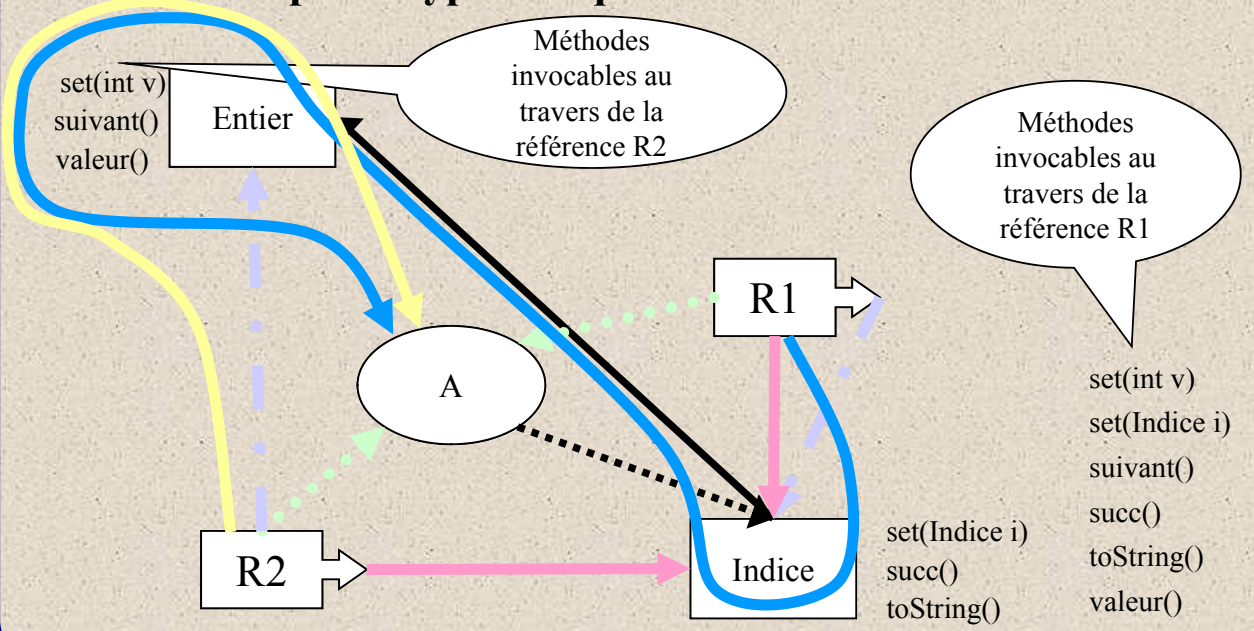
- le type de la référence y donnant accès
- le type effectif

- Type statique : celui au travers duquel on veut voir cet objet à un instant donné. Type associé à la référence.
- Type dynamique (le type réel de l'objet accessible au travers de la référence) : celui qui a servi à la création de cet objet
- Le type(classe) dynamique doit être compatible avec le type statique.
  - Cette compatibilité est exprimée par la relation d'héritage.



Polymorphisme & Envoi de message

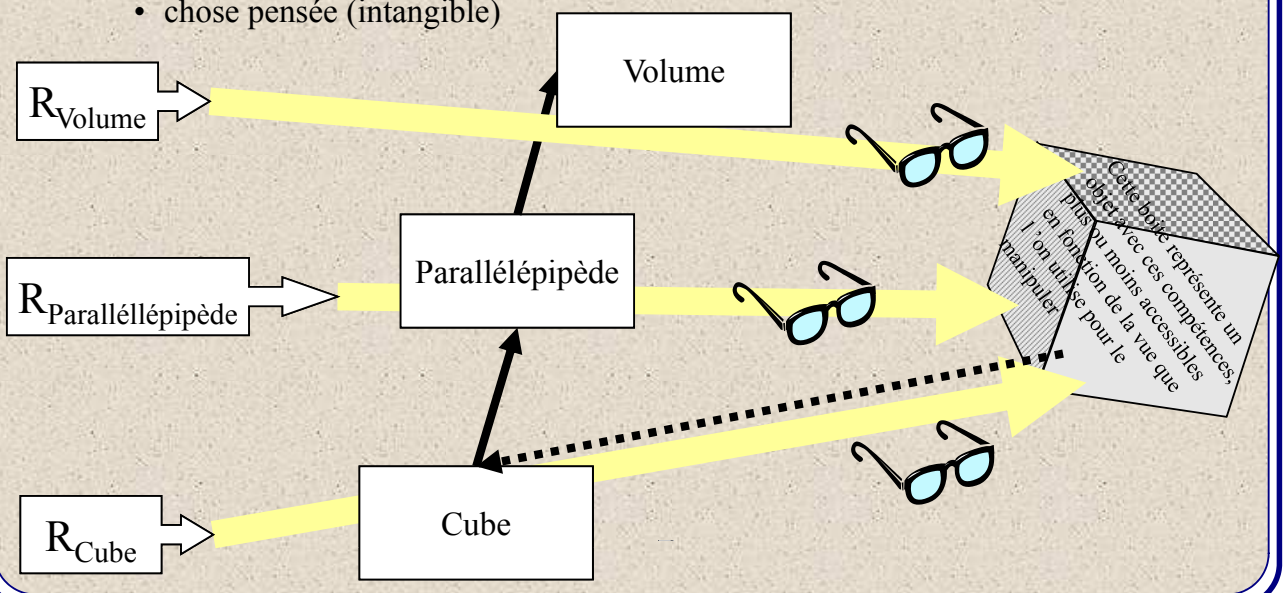
Les messages transmissibles au travers d'une référence sont déterminés par le type statique de cette référence.



Polymorphisme nouménal

Un même objet peut être accédé par des références de types potentiellement différents

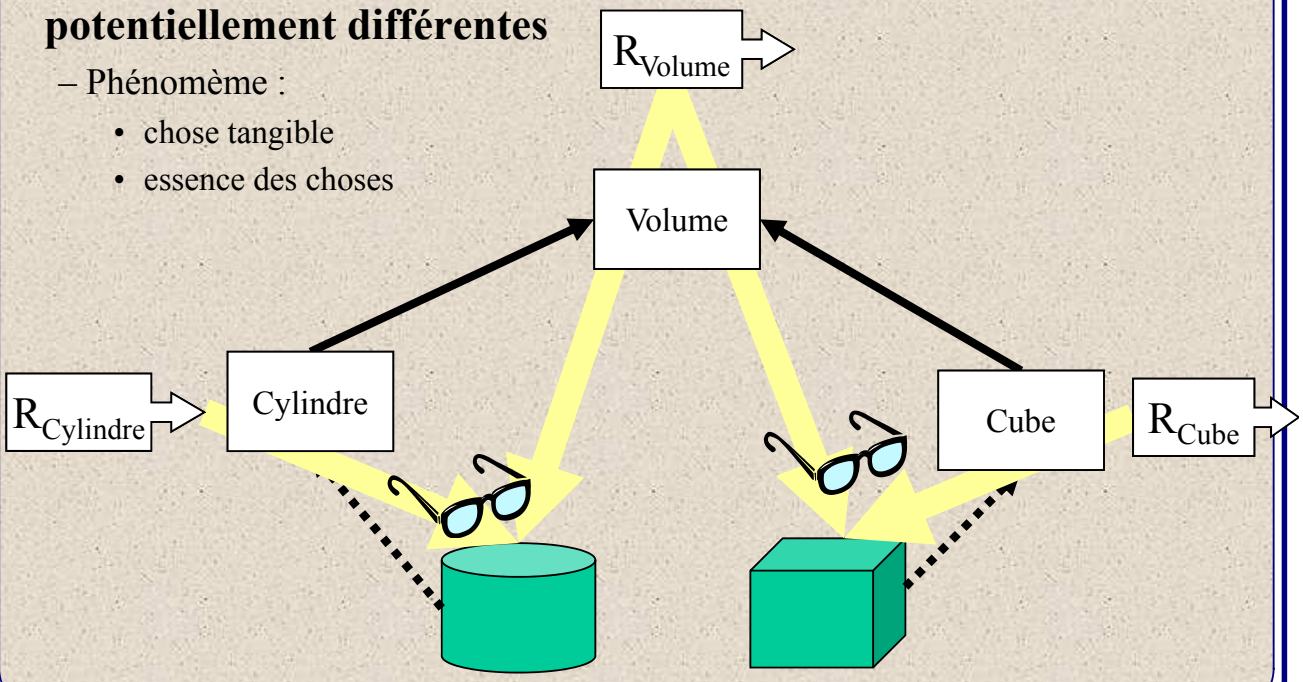
– Noumène
- chose pensée (intangible)



Polymorphisme phénoménal

Une même référence peut donner accès à des objets de natures potentiellement différentes

– Phénomème :
  - chose tangible
  - essence des choses

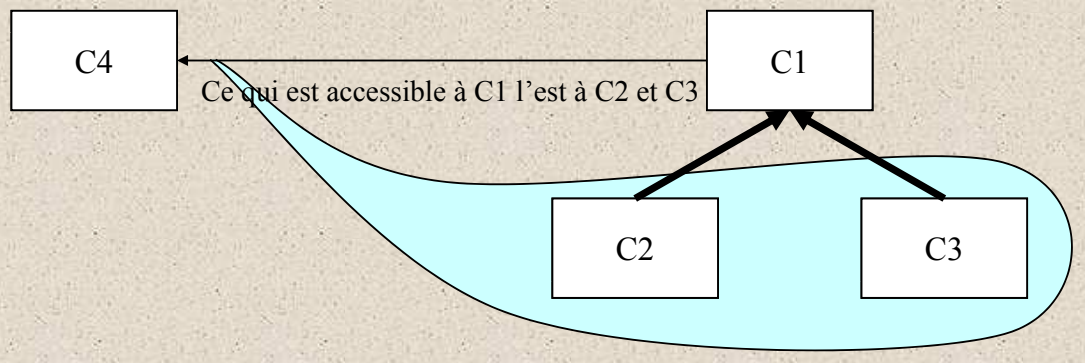


Héritage & Portée

Toute classe doit avoir les privilèges de ses parents

– Toute méthode exportée pour une classe est exportée pour sa descendance.

– On ne peut pas supprimer un droit acquis

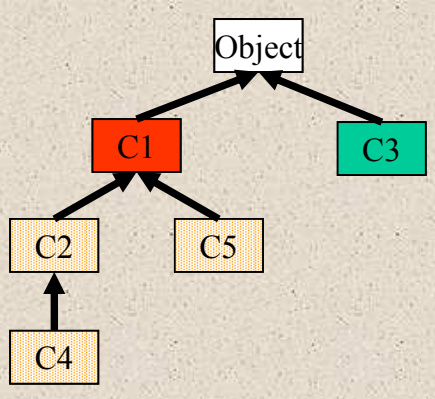


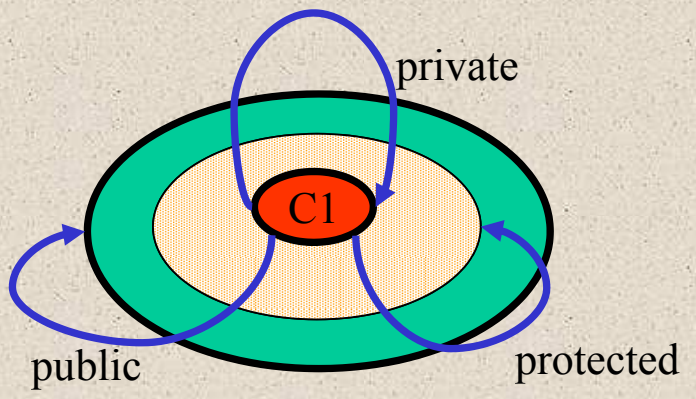
Héritage & Portée

La portée définit l'accessibilité des caractéristiques

- C'est l'expression d'un droit d'accès
  - Par liste d'accès
  - Par groupe

Attention : en Java la protection est réalisée de manière ensembliste

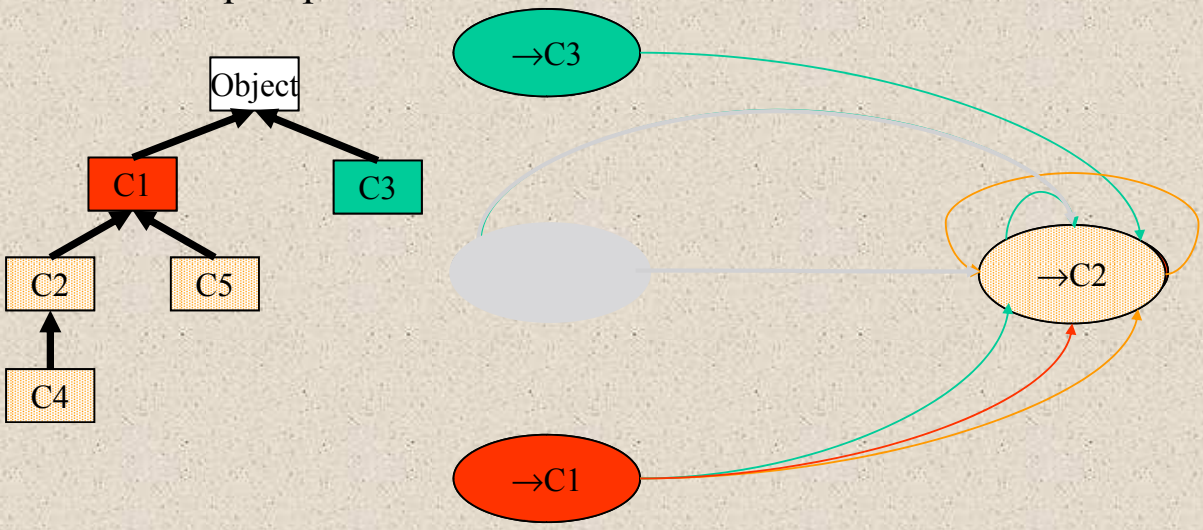




Héritage & Portée

L'encapsulation n'est pas garantie

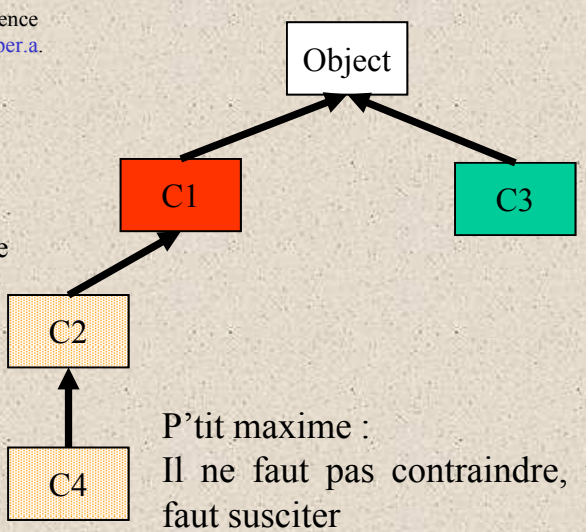
Le paradoxe est qu'un congénère a plus de droits qu'un héritier. Un objet d'une classe dérivée ne peut accéder à une de ses propres caractéristiques privées héritées



Héritage & Portée

HERITAGE DE L'INTERFACE

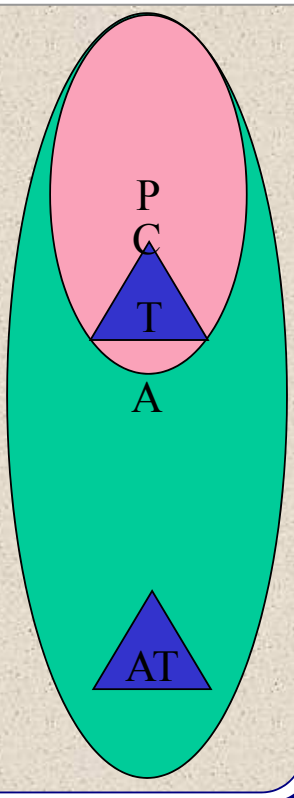
- Le statut d'export des caractéristiques est automatiquement hérité.
- Attention à la redéfinition avec une autre
  - en java la modification d'une portée sur une caractéristique héritée
    - attribut ⇒surcharge
      - » La surcharge est systématique, on peut faire référence à l'attribut "a" surchargé de la classe mère par super.a.
    - méthode ⇒accepté si < sinon erreur
      - » Public
      - » Protected
      - » Package
      - » Private
  - Pour les méthodes on dira que ce mécanisme est contra-variant (voir plus loin)
    - On ne peut pas réduire la visibilité



P'tit maxime :
Il ne faut pas contraindre, il faut susciter

Le cas complet de Java

situation	public	*package*	protected	private protected	private
Accessible à {C}	X	X	X		X
Accessible à {P}	X	X	X		
Accessible à {T}	X	X	X		
Accessible à {A}	X				
Accessible à {AT}	X				
Hérité par {T}	X	X	X	C	
Hérité par{AT}	X		X	X	



Mécanismes de différenciation

Différenciation entre les classes Mère & Fille

– Ajout de compétences (extension)
  - La classe héritant ajoute de nouvelles compétences par rapport à la classe dont elle hérite.
    – ne pose pas de problème de principe

– Elimination de compétences (suppression)
  - La classe héritant supprime des compétences par rapport à la classe dont elle hérite
    – Pose un problème de principe : les objets de cette classe ne peuvent pas assurer le comportement de leurs ascendants.

– Affinement de compétences (redéfinition)
  - On modifie les définitions des compétences héritées de la classe mère par un mécanisme de redéfinition. Ces modifications peuvent être :
    – Structurelle (Nature) : en modifiant le type des compétences
    – Comportementale (Valeur) : en modifiant la valeur des compétences
    – Contractuelle (Propriété) : en modifiant les contraintes sur les compétences

Affinement Structurel

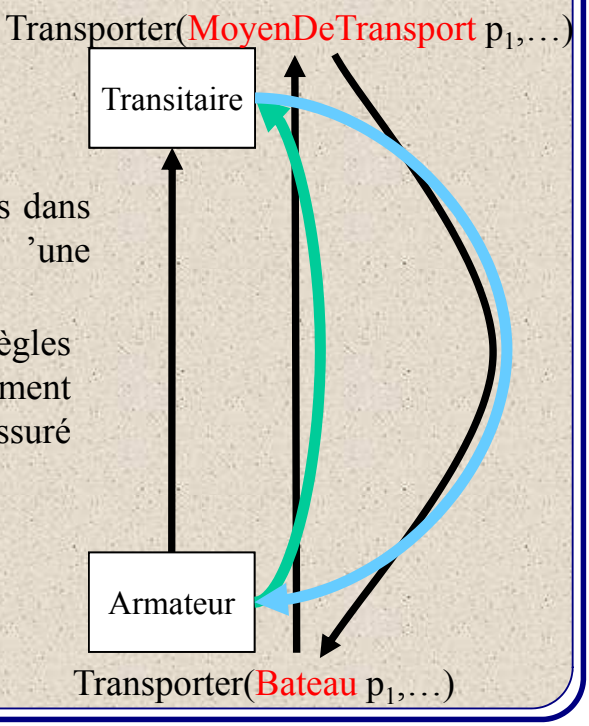
– On modifie les types (classes) d'une ou plusieurs entités héritées par la classe :
  - attribut
  - paramètre
  - résultat de fonction

– Cette possibilité permet d'être plus précis dans la classe fille, puisqu'on dispose d 'une information plus précise.

– Cette modification doit respecter des règles précises. Faire en sorte que le comportement décrit dans la classe mère soit toujours assuré par les instances de la classe fille.
  - Compatibilité co-variante de types
  - Compatibilité contra-variante de types

– Rentre en conflit avec la surcharge

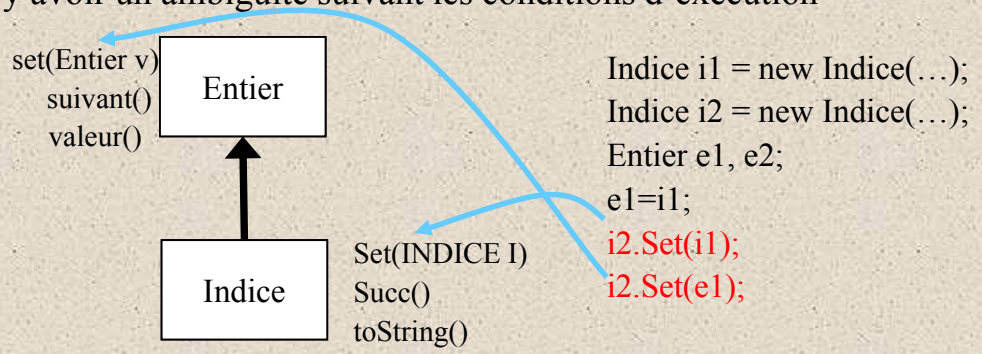
– N'est pas sans problème aussi



Héritage & Surcharge

La surcharge permet d'avoir dans le dictionnaire d'une classe 2 caractéristiques portant le même nom.

- Soit il y a masquage (cas des attributs)
- Soit la distinction est faite sur critère de sélection plus large (types des paramètres pour le cas des méthodes)
- Il peut y avoir un ambiguïté suivant les conditions d'exécution



La propriété d'appartenance à l'intervalle n'est pas garantie car la méthode Set de Entier peut être exécutée. Pour être correcte, la classe Indice devrait aussi redéfinir la méthode set(Entier)

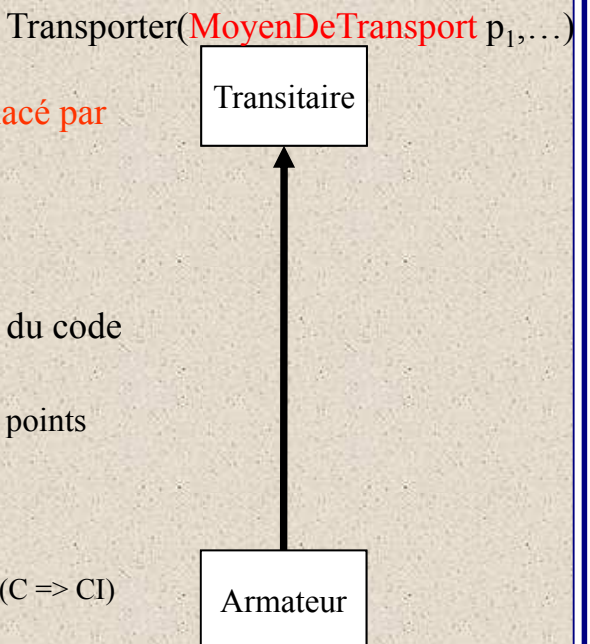
Affinement Comportemental

Remplacement

- Le code de la procédure héritée est remplacé par un nouveau code.
- la valeur d'un attribut hérité est fixée

Adjonction

- Le code de la procédure est complété par du code spécifique à la classe fille.
  - Les adjonctions (AD) interviennent en des points précis du code initial (CI)
    - début (AD;CI)
    - fin (CI;AD)
    - ou le code initial est exécuté sous condition (C => CI)



ENTIER

```
class Entier {
    public Entier(int valeur) {set(valeur);
    }
    /**
    * @fonction: affecte l'entier avec la valeur du paramètre
    */
    public void set(int valeur) {this.valeur = valeur;
    }
    /**
    * @fonction: fait progresser l'entier d  'une unité
    */
    public void suivant() { valeur++;
    }
    /**
    * @fonction: restitue la valeur courante de l'entier
    */
    public int valeur() {return valeur;
    }
    protected int valeur;
}
```


INDICE

```
public class Indice extends Entier {
    /**
    * @fonction: création d'un indice dans l'intervalle [binf..bsup]
    */
    public Indice(int binf,bsup) {this.binf=binf; this.bsup=bsup;super.set(binf);
    }
    /**
    * @fonction: affecte l'indice avec la valeur de l'indice argument. celle-ci doit
    *    respecter les bornes
    */
    public void set(Indice i) {set (i.valeur());
    }
    /**
    * @fonction: fait progresser l'indice d'une unité s'il n'est pas sur bsup
    */
    public void suivant() {if(valeur  < bsup) valeur++;
    }
    /**
    * @fonction: retitue la représentation textuel de l'objet
    */
    public String  toString() {
        return '['+binf+'/'+valeur+'/'+bsup+']';
    }
    // Valeur et bornes de l'indice.
    private int binf, bsup;
}
```


Redéfinition : liaison dynamique

Tout objet exécute la plus proche sémantique du message qu 'il reçoit

- Soit une méthode capacite définie dans Volume dont la sémantique est $capacite^{Volume}$. On dit que $capacite_{(Volume)} \equiv capacite^{Volume}$
- La méthode capacite est redéfinie dans Cylindre et Cube avec les sémantiques respectives $capacite^{Cylindre}$ et $capacite^{Cube}$
- On suppose que $R_{Cylindre}$ et R_{Cube} donnent accès à des objets de type Cylindre et respectivement Cube.
- $R_{Cylindre}$•capacite ↙$capacite^{Cylindre}$
- R_{cube}• capacite ↙ $capacite^{Cube}$

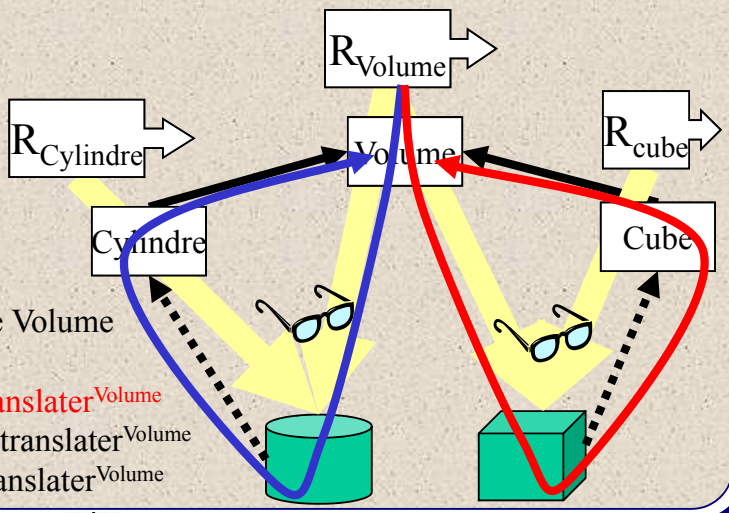
- $R_{Volume} = R_{Cylindre}$;
  R_{Volume}• capacite ↙ $capacite^{Cylindre}$
- $R_{Volume} = R_{cube}$;
  R_{Volume}•capacite ↙ $capacite^{Cube}$

Si $translater_{(Volume)}$ est une méthode de Volume non redéfinie dans Cylindre et Cube

$translater_{(Cylindre)} = translater_{(Cube)} \equiv translater^{Volume}$

$R_{Volume} = R_{Cylindre}$; R_{Volume}•translater ↙$translater^{Volume}$

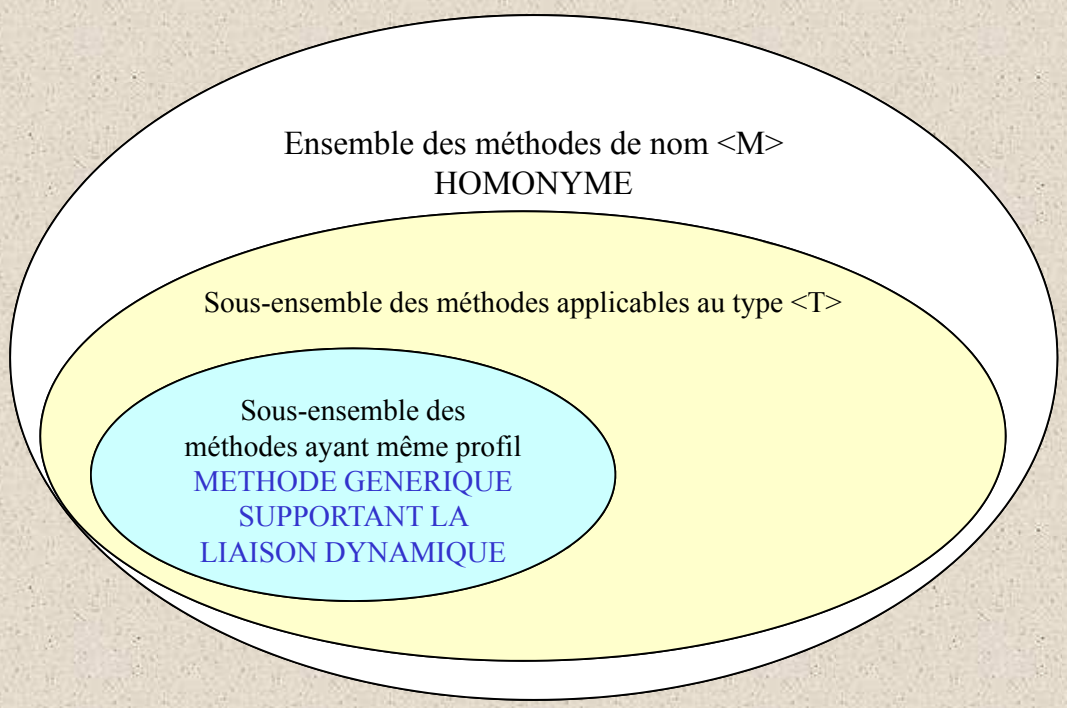
$R_{Volume} = R_{cube}$; R_{Volume}•translater ↙ $translater^{Volume}$



Redéfinition vs Surcharge

		Surcharge → Liaison statique		
Classe C ↑	Redéfinition — Liaison dynamique	$<T_x>$ M($<D_1>$)		
Classe C2		$<T_x>$ M($<D_1>$)	$<T_x>$ M($<D_2>$)	
Classe C3				$<T_x>$ M($<D_i>$)

Liaison dynamique



Redéfinition structurelle en Java

On écrit dans B une surcharge et une redéfinition de M. 2 cas se présentent pour l'écriture de la redéfinition :

– M est totale en toute circonstance

```
M(T1 p1,…) {
    if(p1 instanceof T2 && …)  M((T2)p1, …);
    else {
        <traitement spécifique>
    }
}
```

– M est partielle, une exception est émise en conséquence

```
M(T1 p1,…) throws RuntimeException {
    if(p1 instanceof T2 && …)  M((T2)p1, …);
    } else  throw new RuntimeException( …);
}
```

Affinement Contractuel

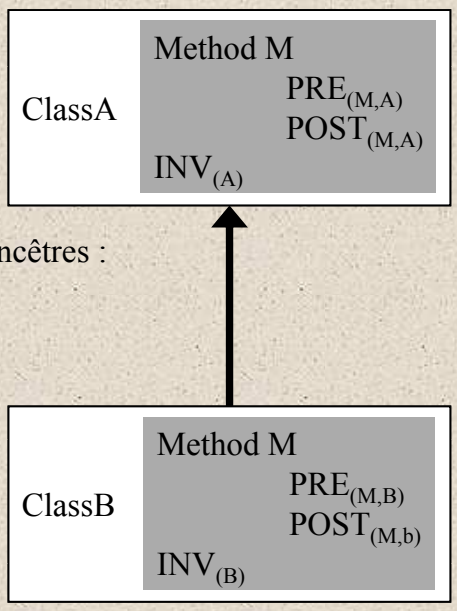
Les contraintes générales
- invariant de classe

les contraintes associées à une méthode
- pre-condition
- post-condition

L 'héritage doit conserver le contrat
- Les descendants doivent assurer les contrats de leurs ancêtres :

```
ClassA a = new ClassB();
a.M();
```

- Statiquement le contrat est :
  - $C_{(M,A)} = PRE_{(M,A)} \{DO_{(M,A)}\} POST_{(M,A)}$
- Dynamiquement le contrat est :
  - $C_{(M,B)} = PRE_{(M,B)} \{DO_{(M,B)}\} POST_{(M,B)}$
- Il faut que
  - $PRE_{(M,A)} \Rightarrow PRE_{(M,B)}$ affaiblissement
  - $POST_{(M,B)} \Rightarrow POST_{(M,A)}$ renforcement



Héritage & Assertion

Par application de la logique propositionnelle :
- $\lambda \Rightarrow (\lambda \vee \mu)$ règle d'introduction
- $(\lambda \wedge \mu) \Rightarrow \lambda$ règle d'élimination

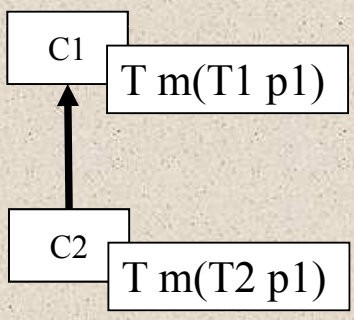
Les PRE&POST conditions sont ;
- $PRE_{(M,B)} = PRE_{(M,A)} \vee PRE\_ELSE_{(M,B)}$
- $POST_{(M,B)} = POST\_THEN_{(M,B)} \wedge POST_{(M,A)}$

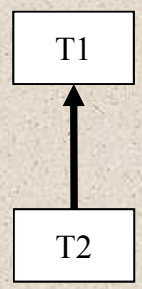
Les invariants de classe sont aussi hérités ("et" logique)
- $INV_{(B)} = INV\_THEN_{(B)} \wedge INV_{(A)}$

Rq :
- $PRE\_ELSE_{(M,B)}$ absente $\Leftrightarrow PRE\_ELSE_{(M,B)}$ = false
- $POST\_THEN_{(M,B)}$ absente $\Leftrightarrow POST\_THEN_{(M,B)}$ = true
- $INV\_THEN_{(B)}$ absente $\Leftrightarrow INV\_THEN_{(B)}$ = true
- PRE, POST, INV absentes $\Leftrightarrow$ PRE,POST,INV = true

Héritage & Redéfinition

Problème lié à l'affinement structurel co-variant






```
C1 o1 = new C2();
T1 t1 = new T1();
o1.m(t1);
```

Statiquement que peut-on conclure ?

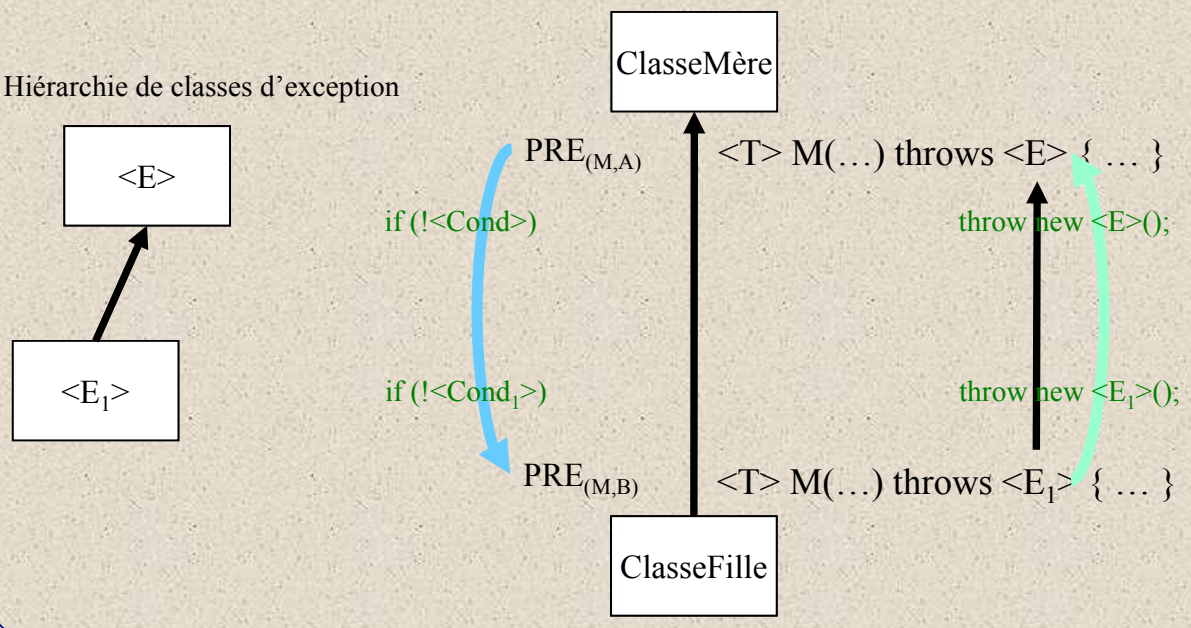
Dynamiquement que peut-on dire ?

co-variant
contra-variant
in-variant

Héritage & Exception

Le principe de l'affinement des exceptions est co-variant, c-a-d que le raffinement M' d'une méthode M ne peut transmettre qu'un type E1 d'exception dérivé du type E transmis par M. Ceci ne concerne les RuntimeException et Error.

Hiérarchie de classes d'exception



Héritage & Compatibilité

La compatibilité de profils suit les règles suivantes :

Soit M une méthode dont le profil est le suivant :

- $<T_0>$ <M>($<T_1>p_1,\ldots, <T_m>p_m$) throws $E_1, \ldots, E_n$

Ce profil est équivalent au profil suivant par la règle qu'une exception appartenant à la hiérarchie de RuntimeException n'est pas obligatoirement spécifiée

- $<T_0>$ <M>($<T_1>p_1,\ldots, <T_m>p_m$) throws $E_1, \ldots, E_n$, RuntimeException

Le profil suivant :

- $<T_0'>$ <M>($<T_1'>p_1,\ldots, <T_m'>p_m$) throws $E_1', \ldots, E_k'$

Est compatible au profil suivant :

- $<T_0>$ <M>($<T_1>p_1,\ldots, <T_m>p_m$) throws $E_1, \ldots, E_n$, RuntimeException
  - Si $\forall i \in \{1,m\}\ T_i' \subseteq T_i \wedge \forall j \in \{1,k\}, \exists r \in \{1,n\}\ E_j' \subseteq E_r \vee T_j' \subseteq$ RuntimeException
  - Si $T_0' \subseteq T_0$

Une méthode M de profil P' appartenant à la classe C' redéfinit une méthode M de profil P appartenant à la classe C si C' hérite de C et le profil P' est compatible au profil P et les types des paramètres sont les mêmes $T_i = T_i'$.

- public Entier successeur() {return new Entier(valeur+1);}
- public Indice successeur() {Indice i =new Indice(binf,bsup); i.set(valeur+1); return i;}

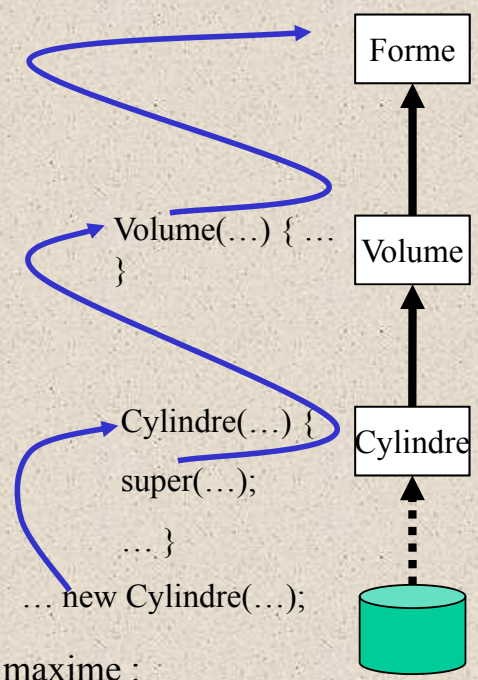
Héritage & Constructeurs

L'héritage n'est pas la composition ; construire un objet ne se réduit pas construire ses formes ancestrales.

- Ce postulat est d'ailleurs celui appliqué par la redéfinition pour les méthodes classiques.
- C'est aussi conforté, d'une certaine manière, par l'existence de multiples créateurs qui sont autant de façons différentes de créer un même objet.

Cependant en Java comme en C++ ce n'est pas la règle

- Champ private



P'tite maxime :

Il est préférable d'utiliser une expression que du code pour exprimer un invariant

Héritage & Constructeurs

Pour réduire ce problème lié au constructeur :

– On réduit le corps du constructeur à l'appel d'un « initialisateur »

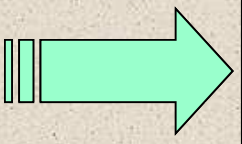
– Cet « initialisateur » peut être redéfini, ce qui permet l'apdatation

```
class X {

    public X(<T> p*){ init(p*);}

    protected init<T>p*){ …};
}
class T extends X {

    public Y(<T> p*){super(p*);}

    protected init(<T>p*){ …; super.init(p*); …}
}
```


Héritage vs Délégation

On peut, dans une certaine mesure, remplacer un héritage par une délégation

```
public class C1 {
    public T1 A1;
    public void M(…) …
    private T2 A2;
}
public class C2 extends C1 {
    …
}
```



```
public class C2 {
    private C1 H;
    public T1 A1() { return H.A1;}
    public void A1(T1 p) {H.A1=p;}
    public void M(…) H.M();}
}
```

Remarques :

Accroît le code

La portée « protected » n'a plus de sens

Interdit toutes les possibilités liées à l'héritage !

soit on considère que l'héritage est inutile, auquel cas !!!

soit les possibilités déjà énoncées paraissent essentielles

Héritage vs Délégation

Le modèle C2 hérite du modèle C1 quand :

- Les objets de C2 sont des C1
  - $\forall C, C \in C2 \Rightarrow C \in C1$ ($C2 \subseteq C1$)
- Les objets de C2 ont les propriétés de C1
  - $\forall C, C2(C) \Rightarrow C1(C)$

Le modèle C2 délègue au modèle C1 quand :

- Les objets de C2 utilise les services de C1

Les 2 formes ne sont pas exclusives

- Une classe peut à la fois hériter d'une classe et être cliente de cette même classe.

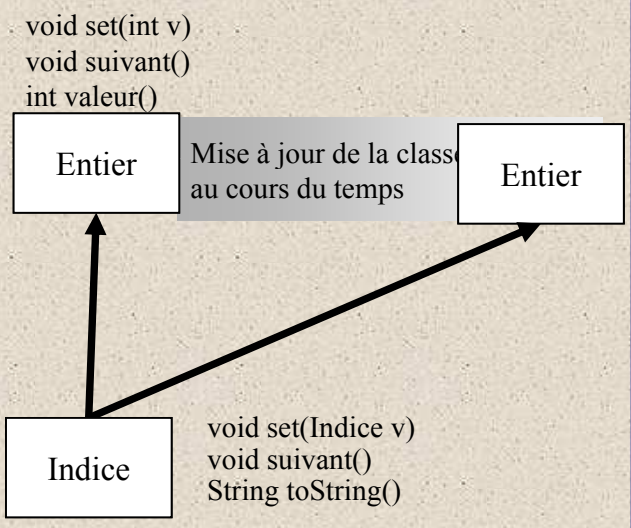
Utiliser l'héritage autant que faire se peut !

Héritage & Evolutivité

Le mécanisme associé au concept d'héritage doit permet de maintenir une cohérence lors de la mise en place de modifications ou d'évolutions.

- Une classe héritière doit être écrite en termes des compétences du niveau d'abstraction «supérieur», et ne doit en aucun cas être écrite en termes des réalisations du niveau d'abstraction «supérieur ».
- A l'extrême on crée une parfaite étanchéité entre les 2 classes. Tout attribut dans la classe mère est privé et des méthodes protected de la classe mère permettent la manipulation depuis la classe fille.

Ou plaidoyer pour les « private » ?!



P'tite maxime :

L'extrémisme n'est jamais une solution universelle.

INDICE

```
public class Indice extends Entier{
    /**
    * @fonction: création d'un indice dans l'intervalle [binf..bsup]
    */
     public Indice(int binf,bsup) {this.binf=binf; this.bsup=bsup;super.set(binf);
     }
    /**
    * @fonction: affecte l'indice avec la valeur de l'indice argument. celle-ci doit
    *    respecter les bornes
    */
    public void Set(Indice i) {set (i.valeur());
    }
    /**
    * @fonction: fait progresser l'indice d'une unité s'il n'est pas sur bsup
    */
     public void suivant() {if(valeur  < bsup) valeur++;
    }
    /**
    * @fonction: restitue la représentation textuel de l'objet
    */
    public String  toString() {
        return '['+binf+'/'+valeur()+'/'+bsup+']';
    }
    // Valeur et bornes de l'indice.
     private int binf, bsup;
}
```

super.suivant();