M1 info

GINF41B2 (Conception et Programmation Orientée Objet)

Cours #5
Polymorphisme

Pierre Tchounikine

Plan

- Héritage, typage et transtypage (retour sur)
  - *up casting, down casting*
- *Mécanisme de liaison*
  - *early binding, late binding*
- Remplacement (≠ surcharge)
- Polymorphisme

Héritage, typage et transtypage
(retour sur)

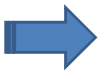
Héritage et typage

- Un objet peut être de plusieurs types
- Le typage a une dimension dynamique

il y a 2 types !

  - le type à travers lequel on voit l'objet à un moment donné, associé à **la référence** à l'objet qui est utilisée
  - le type réel de l'objet, celui avec lequel il a été **créé**

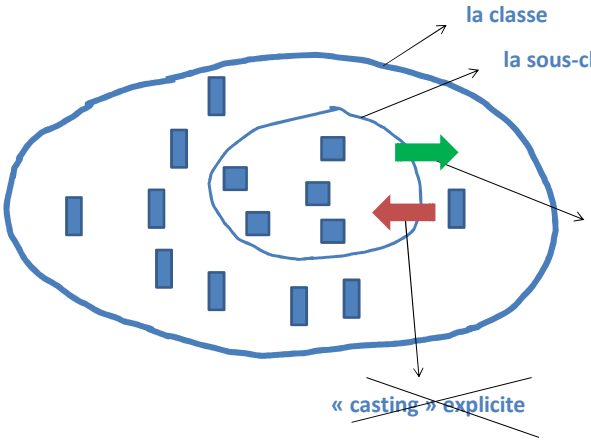
- **Difficulté** : faire attention à la compatibilité des types que définit l'héritage
- **Avantage** : possibilité de manipuler des objets « différents » comme des objets de même type (à un certain niveau d'abstraction)



polymorphisme et utilisation du principe de « liaison dynamique »

Héritage : vision ensembliste

- Les instances de la sous-classe forment un sous-ensemble des instances de la super-classe



les objets de la sous-classe sont (aussi) des objets de la classe

Up-cast et Down-cast

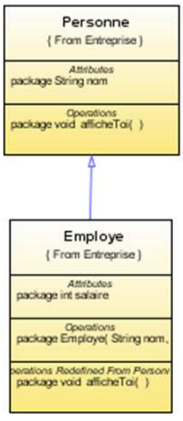
Trans-typage

Up-casting

voir les objets d'une sous-classe comme des éléments de sa super-classe

un employé est une personne par définition

toute méthode de la super-classe est aussi une méthode de la sous-classe dérivée



Down-casting

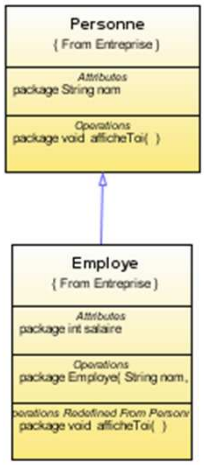
voir les objets d'une sur-classe comme des éléments de sa sous-classe

une personne peut être un employé (ou pas)

la sous-classe étend la super-classe et peut avoir des méthodes que la super-classe n'a pas

Héritage et down-casting (1/3)

Principe de substitution
on peut « remplacer » un objet de la classe mère par un objet de la classe fille

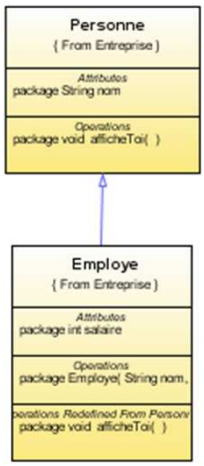



```
Personne p;
Employe e;
p=e;  // légal ?  → oui, un employé est une personne
e=p; // légal ?  → non, une personne n'est pas un employé !
```

```
Personne a=new Employe(« titi », 3); // légal ?  → oui, un employé est une personne
Employe b=new Personne(); // légal ?  → non, une personne n'est pas un employé !
```

(instance « a » créée comme un Employé)

Héritage et down-casting (2/3)




```
Personne p=new Personne();
p.nom="tutu";          → à éviter, encapsulation !
Employe e;
//p=e;  licite
//e=p;  illicite
e=(Employe)p;   // légal?
```

Compilation : OK (ça aurait pu être un Employé …)

Exécution : Exception in thread "main"
java.lang.ClassCastException:
Personne cannot be cast to Employe

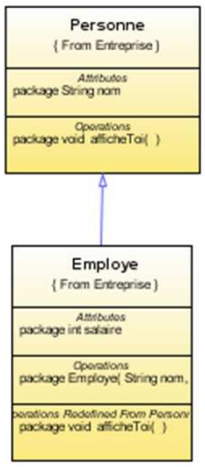
(… mais c'est une Personne ☹)

un down-cast illégal provoque ClassCastException à l'exécution

Héritage et down-casting (3/3)

```
Personne p =new Personne();
System.out.println(p.toString());
((Employe)p).salaire=10;  // légal ?
```

→ **Personne@19821f**



Compilation : OK (p pourrait être un Employé …)

Exécution : Exception in thread "main" java.lang.ClassCastException: Personne cannot be cast to Personne

(… mais c'est une Personne et une Personne n'a pas de salaire ☹)

un down-cast illégal provoque ClassCastException à l'exécution

si on veut vraiment down-caster sans risque : if (x instanceof Classe) …

9/46
Cours P. Tchounikine

Mécanisme de liaison

10/46
Cours P. Tchounikine

Liaison dynamique (retour sur l'exemple)

```
class Personne {
String nom;
void afficheToi(){
System.out.println(nom);}
}
class Employe extends Personne {
int salaire;
Employe(String nom, int salaire) {this.nom=nom; this.salaire=salaire;}
void afficheToi(){System.out.println(nom + "  " + salaire);}
void augmenteToi (int val){salaire=salaire+val;}
}
```

```
Personne p = new Personne();
p.nom="titi";
p.afficheToi();          → titi
Employe e = new Employe("tutu", 2000);
e.augmenteToi(500);
p=e;
p.afficheToi();          → tutu 2500
```

c'est le « afficheToi » du type effectif de p (de l'objet référencé par p) qui est déclenché

Mécanismes de liaison

ligature dynamique, late binding

- **Liaison statique** :

  le code de la fonction à exécuter est déterminé à la compilation

- **Liaison dynamique :**

  le code de la fonction à exécuter est déterminé à l'exécution (run time)

  – le compilateur vérifie que la fonction (la méthode / le message) existe
  → signature
  – la méthode effective est déterminée (par la JVM) à l'exécution, en fonction du type effectif de l'objet concerné

Liaison dynamique : détails

étant donné un appel objet.méthode()

- A la compilation :
  - détermination de la signature de la méthode convenant le mieux à l'appel
    - en partant de la classe de l'objet
    - en remontant la hiérarchie éventuellement
- A l'exécution :
  - détermination de la méthode de signature et de type de retour voulus (ou co-variant *cf. plus tard*)
    - à partir du type effectif de l'objet référencé
    - en remontant dans la hiérarchie éventuellement

Héritage et typage (résumé)

- Un objet peut être de plusieurs types (héritage)
- Le typage a une dimension dynamique
  - le type à travers lequel on voit l'objet à un moment donné, associé à la **déclaration de la référence** à l'objet qui est utilisée

    détermine les messages transmissibles à l'objet
  - le **type réel** de l'objet, celui avec lequel il a été **créé**

    détermine le comportement effectif de l'objet
- Une même référence peut donner accès à des objets de types différents (impliqués dans une relation d'héritage)
- La liaison est dynamique → sur l'objet effectif qui reçoit le message

si pas de bidouille (genre down-cast), pas de risque

Redéfinition des méthodes

Redéfinition

override

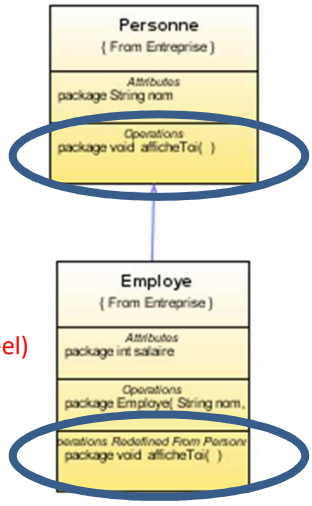
- Idée :
  - la sous-classe propose une méthode qui redéfinit (qui donne une définition locale) d'une méthode d'une classe ascendante
- Conséquences
  - la méthode de la sous-classe **masque** (va être exécutée à la place) de la méthode de la classe ascendante
- Utilité
  - possibilité d'avoir des objets d'une même sur-classe qui ont des comportements communs mais différents
- Base syntaxique
  - même « signature » (types valeurs d'entrée) **+** même valeur de retour

Redéfinition : retour sur l'exemple

```
Personne p = new Employe("toto",4);
p.afficheToi();
```

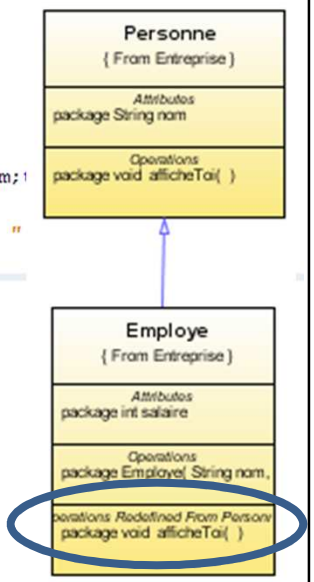
toto 4 (déclenche le afficheToi de Employé = type réel)

toto est un Employe
et
Employe a une méthode afficheToi qui redéfinit celle de Personne

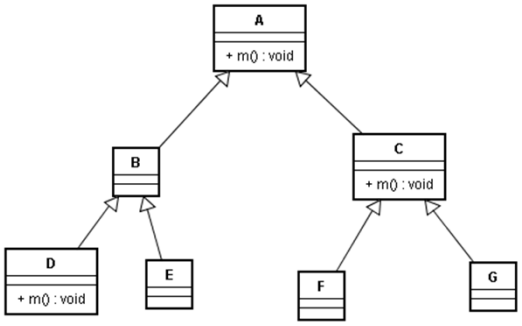


Re-définition

```
class Employe extends Personne {
int salaire;
Employe(String nom, int salaire) {this.nom=nom;
    @Override
void afficheToi(){System.out.println(nom + "  "
}
```



Redéfinition : plusieurs niveaux



- Au niveau de A : accès à la méthode m de … **A**
- Au niveau de B : accès à la méthode m de … **A**
- Au niveau de C : accès à la méthode m de … **C**
- Au niveau de D : accès à la méthode m de … **D**
- Au niveau de E : accès à la méthode m de … **A**
- Au niveau de F : accès à la méthode m de … **C**

Surcharge

ne pas confondre

sur-définition, overload

- Idée
  - un même symbole possède plusieurs significations
  - la signification utilisée est choisie en fonction du contexte

3+4, 3.1 + 3.2, etc

- Conséquences (dans le cas de la surcharge de méthodes)
  - plusieurs méthodes de même nom (mais de profils différents)
- Utilité :
  - éviter la multiplication des identificateurs
  - faciliter l'utilisation des classes / des bibliothèques
- Base syntaxique
  - un même nom
  - des « signatures » (types valeurs d'entrée) différentes

→ une possibilité de différenciation

exemple : proposer plusieurs constructeurs

Surcharge et redéfinition

- La surcharge
  - une notion générale
  - qui peut être associée à une situation d'héritage

`paramètres d'entrée différents`
`type de retour sans importance`

interface modifiée : ajout d'un comportement

- La redéfinition
  - une notion liée à la notion d'héritage

`identité des « signatures »`
`et du type de retour`

(même type ou co-variant, cf. plus tard)

interface de la classe inchangée
implantation de la classe modifiée

surcharge : permet de cumuler des méthodes de même nom
redéfinition : substitue une méthode à une autre

Surcharge et redéfinition

```
class Employe extends Personne {
int salaire;
Employe(String nom, int salaire) {this.nom=nom; this.salaire=salaire;}
void afficheToi(){System.out.println(nom + "  " + salaire);}
void augmenteToi (int val){salaire=salaire+val;}
void augmenteToi (float val){salaire=(int)(salaire+salaire*val);}}

class Trader extends Employe {
private int tauxdecom;    // très private !
Trader(String nom, int salaire, int tauxdecom) {
      super(nom,salaire);this.tauxdecom=tauxdecom;}
void augmenteToi (int val, int prime){
    salaire=salaire+val+prime;} //prime officielle
void augmenteToi (int val){
    salaire=salaire+val+salaire*tauxdecom/100;} //commission occulte
}
```

```
Employe e= new Employe("Jacques", 10000);
Trader t1= new Trader ("Joe",10000,10);
Trader t2= new Trader ("Bill",10000,10);
e.augmenteToi(2000);
t1.augmenteToi(2000, 500);
t2.augmenteToi(2000);
```

surcharge

redéfinition

Jacques 12.000 = 10.000 + 2.000

Joe 12.500 = 10.000 + 2.000 + 500

Bill 13.000=10.000 + 2.000 + 10% de 10.000

Surcharge et redéfinition

```
class A {
public int methode(int i) {…}
}

class B extends A {
public float methode(int i) {…}
}
```

surcharge ou redéfinition ?

ni l'un ni l'autre : provoque une erreur !

```
methode(int) in B cannot override methode(int) in A;
attempting to use incompatible return type
found   : float
required: int
```

ce n'est pas de la surcharge : les arguments sont les mêmes
ce n'est pas de la redéfinition : les types de retour sont différents

Redéfinition : utiliser la sur-classe

- Schéma récurent dans les cas de redéfinitions :
  - la sur-classe définit une méthode *m*
  - la sous-classe redéfinit *m* localement en
    - appelant le *m* de la surclasse
    - « complétant » le traitement

problème : appel récursif !

```
sur-classe
    void m (paramètres)
{…}

sous-classe
    void m (paramètres) {
          // appel de m
          // compléments
}
```

Maman et fifille

- pour appeler une méthode de la super-classe, on doit précéder le nom de la méthode par le mot clé super : super.m()
- on ne peut pas appeler une méthode de la super-super-classe, i.e., on ne peut pas faire super.super. m()

Redéfinition et visibilité

- On ne peut pas redéfinir une méthode et la rendre moins accessible
  (par exemple, une méthode publique ou protégée de la surclasse ne peut pas devenir privée)

pourquoi ? **par respect du principe de substitution !**

Principe de substitution
on peut « remplacer » un objet de la classe mère par un objet de la classe fille

(si c'était possible, un objet de la classe fille ne remplirait pas le contrat défini par la classe mère ; dans l'autre sens, la sous-classe étend les fonctionnalités de la sur-classe : pas de problème ; et si les attributs de la surclasse sont privés ils ne sont pas accessibles → encapsulation respectée)

- Une méthode redéfinie peut en revanche être rendue plus accessible
  (de private à « package » par exemple)

Redéfinition et covariance

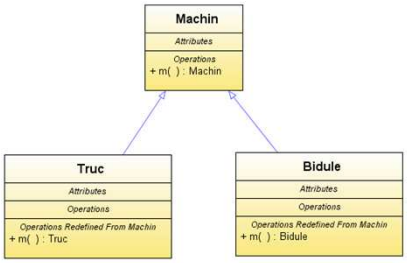
- La surcharge

`paramètres d'entrée différents`
`type de retour sans importance`

- La redéfinition
  - une notion liée à la notion d 'héritage

valeurs de retour **covariantes**
(même type ou type dérivé ; >JDK 5.0)

`identité des « signatures »`
`et ~~du type de retour~~`



car mêmes arguments (ici aucun) !

Truc.m et Bidule.m redéfinissent Machin.m
- ❖ m, appliqué à un Machin, renvoie un Machin
- ❖ m, appliqué à un Truc, renvoie un Truc
- ❖ m, appliqué à un Bidule, renvoie un Bidule

Maman et fifille (résumé)

- Une classe fille peut :
  - ajouter des variables
  - ajouter des méthodes
  - surcharger des méthodes
  - redéfinir des méthodes

faire la même chose
ou plus
ou différemment
mais pas moins

- Une classe fille ne peut pas
  - retirer des champs
  - retirer des méthodes
  - rendre une méthode redéfinie moins accessible

Polymorphisme

Le polymorphisme

- Idée
  - une classe définit un comportement
  - les sous-classes **redéfinissent** ce comportement pour lui donner une sémantique / une réalité conforme à ce que sont (plus précisément) ses objets

  ⇒ tous les objets de la classe et des sous-classes réalisent le comportement (savent réagir au message) **ET** le font selon leur spécificités propres

- Conséquences
  - vu de l'extérieur, on peut adresser un message à un objet sans connaître son type (sa classe) précise

    c'est une autre forme d'encapsulation

- Utilisation
  - demander un service à l'objet sans connaitre son type précis (il suffit de savoir qu'il sait faire)

    c'est une autre forme de délégation

Le polymorphisme : exemples

- Message « affranchir » à un objet postal
  - colis
  - lettre
  - lettre recommandée
  - etc.

  « Le SWICH est à la POO ce que le GOTO est à la programmation structurée »

- Message « démarre » à un feu vert
  - voiture
  - vélo
  - etc.

- Message « click sur la croix » dans une fenêtre
  - fenêtre d'affichage d'un message
  - fenêtre d'une application
  - etc.

Bases techniques

- Compatibilité des types entre une sous-classe et sa sur-classe
  - les objets des sous-classes sont des objets de la sur-classe
  - possibilités de casting 
- Redéfinition des méthodes
  - la sous-classe peut redéfinir les méthodes de la sur-classe (récursivement)
- Liaison dynamique des méthodes
  - la méthode qui va s'exécuter dépend de l'objet réel qui reçoit le message

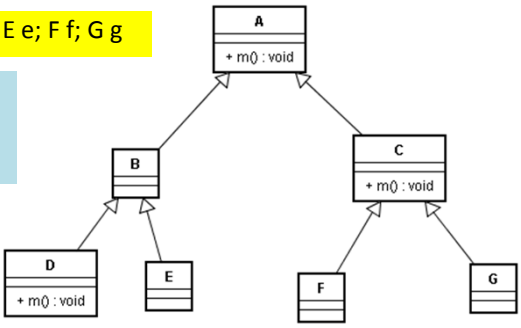
si pas de bidouille (genre down-cast), pas de risque

Polymorphisme : plusieurs niveaux

Déclarations : A a; B b; C c; D d; E e; F f; G g

Affectations légales :
a=b; a=c; a=d; a=e; a=f; a=g;
b=d; b=e; c=f; c=g;

Affectations illégales :
b =a; d=e; d=c; etc.

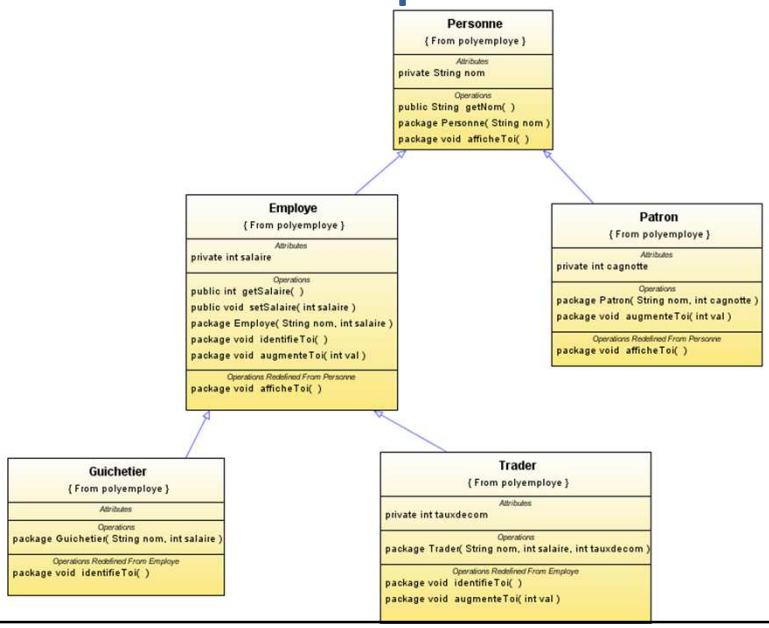


Déclaration : A a

- a référence un A : accès à la méthode m de … **A**
- a référence un B : accès à la méthode m de … **A**
- a référence un C : accès à la méthode m de … **C**
- a référence un D : accès à la méthode m de … **D**
- a référence un E : accès à la méthode m de … **A**
- a référence un F : accès à la méthode m de … **C**

Polymorphisme : exemple

Exemple



La classe Personne

```
class Personne {
private String nom;

public String getNom(){return this.nom;}

Personne (String nom){this.nom=nom;}

void afficheToi(){System.out.println(nom);}

}
```

une Personne a un nom

une Personne peut donner son nom

un constructeur

une Personne sait s'afficher

La classe Employe

```
class Employe extends Personne {
private int salaire;
public int getSalaire()
          {return this.salaire;}
public void setSalaire(int salaire)
          {this.salaire=salaire;}

Employe(String nom, int salaire)
          {super(nom);
          this.salaire=salaire;}

void identifieToi(){System.out.println("Je suis un employé");}

@Override
void afficheToi(){
   System.out.println(getNom() + " Salaire= " + salaire);
   this.identifieToi();}

void augmenteToi (int val){salaire=salaire+val;}
}
```

un Employe est une Personne avec un salaire

un constructeur

un Employe sait s'identifier

on aurait pu utiliser le afficheToi de Personne

un Employe sait s'afficher

un Employe sait s'augmenter

Le Guichetier

```
class Guichetier extends Employe {

Guichetier(String nom, int salaire)
                    {super(nom,salaire);}


@Override
void identifieToi(){System.out.println("Je suis un guichetier");}
}
```

un Guichetier est un Employe

un Guichetier sait s'identifier

NB :

- un Guichetier ne sait pas s'afficher en tant que tel
- un Guichetier ne sait pas s'augmenter en tant que tel

Le Trader

```
class Trader extends Employe {
private int tauxdecom;    // très private !

Trader(String nom, int salaire, int tauxdecom)
      {super(nom, salaire);this.tauxdecom=tauxdecom;}


@Override
void identifieToi(){System.out.println("Je suis un trader");}

@Override
void augmenteToi (int val){setSalaire(getSalaire()+ val +
       getSalaire()*tauxdecom/100);} } //commission occulte
```

un Trader est un Employe avec un taux de com

un Trader sait s'identifier

un Trader sait s'augmenter

NB :

- un Trader ne sait pas s'afficher en tant que tel

Le Patron

```
class Patron extends Personne {
private int cagnotte;

Patron(String nom, int cagnotte){
    super(nom);
    this.cagnotte=cagnotte;}

void augmenteToi(int val)
     {cagnotte=cagnotte+val*10;}

@Override
void afficheToi()
{System.out.println(getNom() + " (et je suis à plaindre je
                                  n'ai pas de salaire)");}

};
```

un Patron est une Personne avec une cagnotte

un Patron sait s'augmenter

NB : un Patron n'est pas un Employe

L'entreprise et ses salariés

```
Personne     p=new Personne("Paul");
Employe      e=new Employe ("Eric", 1000);
Trader       t1=new Trader ("Thierry", 1000, 10);
Trader       t2=new Trader ("Thomas", 1000, 10);
Guichetier   g1=new Guichetier ("Grégoire", 1000);
Guichetier   g2=new Guichetier ("Georges", 1000);
Patron       pat=new Patron("Patrice", 10000);

Personne Entreprise []={t1,g1,t2,g2,e,pat,p};
Employe Salariés []={t1,g1,t2,g2,e};
```

Afficher les membres de l'entreprise

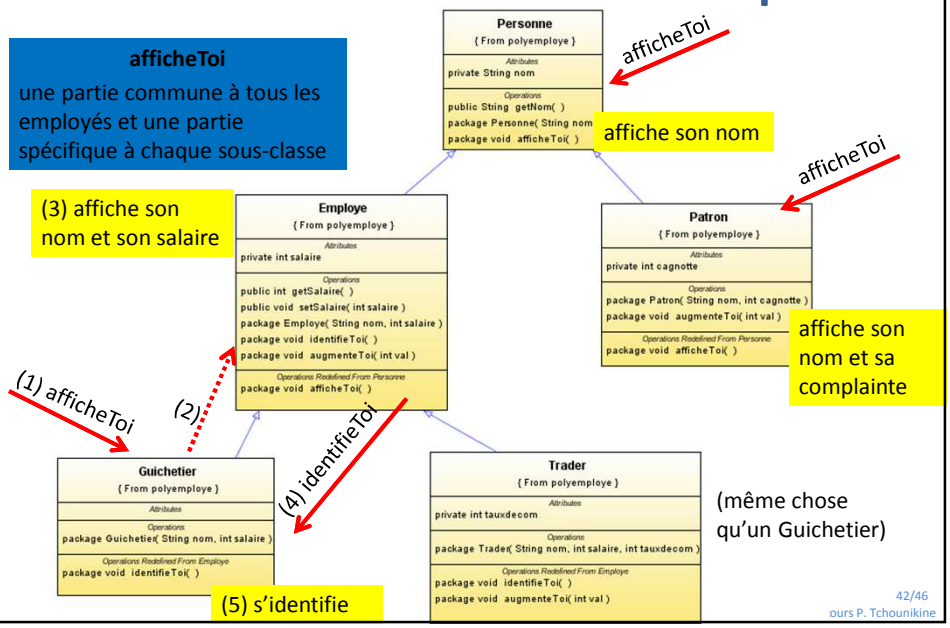
```
Personne Entreprise []={t1,g1,t2,g2,e,pat,p};
Employe Salariés []={t1,g1,t2,g2,e};
```

```
for (i=0;i<7;i++)
    {Entreprise [i].afficheToi();}
```

```
Thierry Salaire= 1000 ; Je suis un trader
Grégoire Salaire= 1000 ; Je suis un guichetier
Thomas Salaire= 1000 ; Je suis un trader
Georges Salaire= 1000 ; Je suis un guichetier
Eric Salaire= 1000 ; Je suis un employé
Patrice (et je suis à plaindre je n'ai pas de salaire)
Paul
```


Afficher les membres de l'entreprise

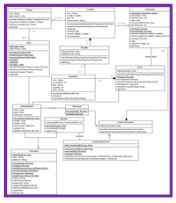
afficheToi
une partie commune à tous les employés et une partie spécifique à chaque sous-classe

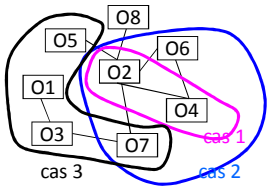


Niveaux modélisation et exécution

rappel

- Le travail de modélisation se fait autour de la notion de classe
- L'exécution du programme est réalisée par les interactions entre objets
  - création d'objets (dynamiquement)
  - interactions entre les objets (envois de messages)





classe
description formelle d'objets ayant une sémantique et des caractéristiques communes

objets
entités discrètes (<identité, état, comportement>) d'un système en cours d'exécution

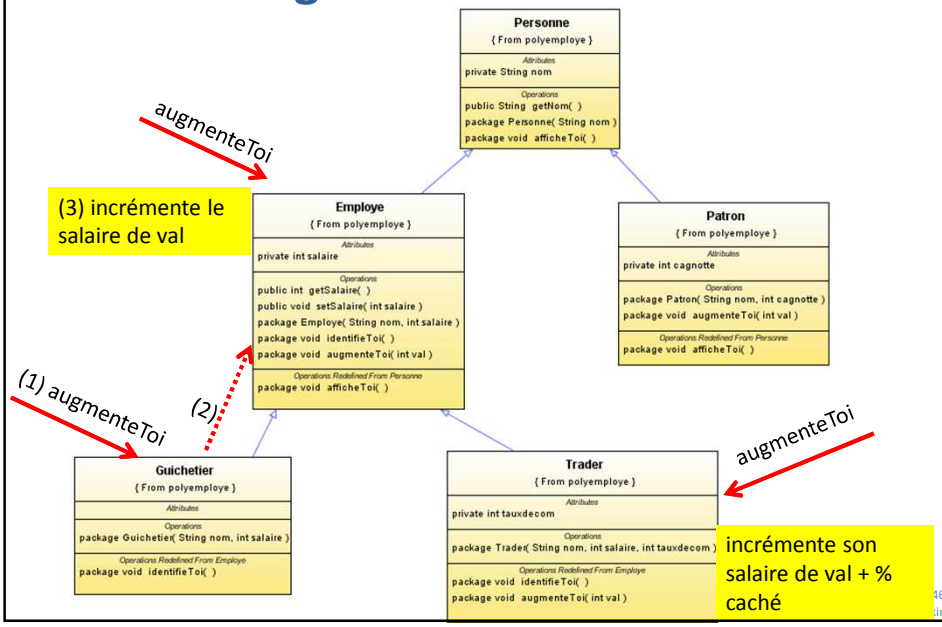
Augmenter les salariés

```
Personne Entreprise []={t1,g1,t2,g2,e,pat,p};
Employe Salariés []={t1,g1,t2,g2,e};
```

```
for (i=0;i<5;i++)
      {Salariés[i].augmenteToi(100);}
for (i=0;i<5;i++)
      {Salariés[i].afficheToi();}
```

```
Thierry Salaire= 1200 ; Je suis un trader
Grégoire Salaire= 1100 ; Je suis un guichetier
Thomas Salaire= 1200 ; Je suis un trader
Georges Salaire= 1100 ; Je suis un guichetier
Eric Salaire= 1100 ; Je suis un employé
```

Augmenter les salariés



Du pas bon et du n'importe quoi

```
Personne Entreprise []={t1,g1,t2,g2,e,pat,p};
Employe Salariés []={t1,g1,t2,g2,e};
```

```
for (i=0;i<7;i++)
   {Entreprise [i].augmenteToi();}
```

erreur de compilation, augmenteToi n'existe pas pour une Personne

```
for (i=0;i<7;i++)
   {((Employe)Entreprise [i]).augmenteToi(100);}
```

erreur d'exécution : un Patron ne peut pas être transtypé en Employe

↓

```
for (i=0;i<5;i++)
    {((Employe)Entreprise [i]).augmenteToi(100);}
```

OK (on a évité le Patron et la Personne) mais moche

erreur ? oui, mais **de modélisation !**

46/46
Cours P. Tchounikine