Comprendre les hiérarchies de types: le polymorphisme en Java

Virginia Aponte

CNAM-Paris

14 mars 2016

Polymorphisme = plusieurs formes (pour les types)

Idée

L'argument passé à une méthode peut avoir *plus d'un type*, si l'algorithme implanté par la méthode ne dépend pas (de manière trop forte) de ce type.

Exemples :

- la méthode `add` qui ajoute un objet en fin de liste, n'utilise pas de connaissance liée au type de l'objet ajouté.

- la méthode qui affiche un objet de type `Affichable`, ne dépend pas du type précis de l'objet, mais seulement du fait qu'il possède une méthode `afficher`;

2 sortes de polymorphisme

- Polymorphisme paramétrique (ou généricité) : méthode « paramétrée » par un type générique T ;
  - Ex : `add(T e)` pour les `ArrayList<T>`.
- Polymorphisme par sous-typage : la méthode est applicable sur tout type « plus bas » dans une hiérarchie de types.
  - Ex : la méthode `aff(Affichable a)` applicable sur tout objet qui implante `Affichable`.

Hiérarchies de types en Java (classes + interfaces)

En Java, les types (classes + interfaces) sont organisées en hiérarchies.
Si B est un « descendant » de A dans une *même* hiérarchie $\Rightarrow$ B *contient au moins* toutes le méthodes de A, plus éventuellement d'autres.

Notation : $A \dashrightarrow B$ siginifie *B descendant de A*.

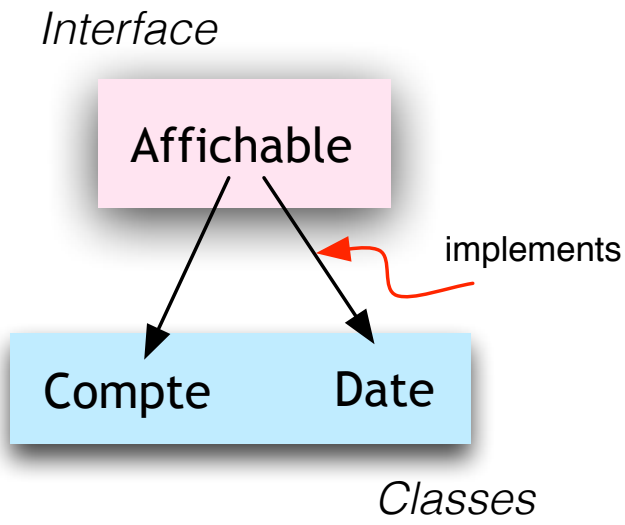
- $I \dashrightarrow C$: entre interfaces et classes via `implements` : si une classe C implante une interface I ;
- $I_1 \dashrightarrow I_2$: entre interfaces via `extends`, si une interface I_2 étend une autre interface I_1 ;
- $C_1 \dashrightarrow C_2$: entre classes via `extends`, si une classe C_2 étend une autre classe C_1 ;

Exemple : hiérarchie entre interfaces et classes

```
interface Affichable {
    void afficher();
}

class Compte implements Affichable {
  …  public void afficher() { …}
}

class Date implements Affichable {
  …. public void afficher() { …}
}
```



Exemple de polymorphisme par sous-typage (1)

```
static void aff(String m, Affichable a){
  Terminal.ecrireString(m);
  a.afficher();
}
public static void main(...){
  Compte c = new Compte();
  Date drv = new Date();
  aff("Etat␣du␣compte", c);
  aff("Date␣du␣rdv", d);
```

`aff` accepte aussi bien un objet Compte qu'un objet Date.

`aff` est polymorphe par sous-typage : elle accepte tout objet dont le type est un descendant du type `Affichable` déclaré pour son argument a.

Exemple : hiérarchie entre classes (1)

`CompteRemunere` *étend* la classe `Compte` avec nouveux attributs + méthodes.

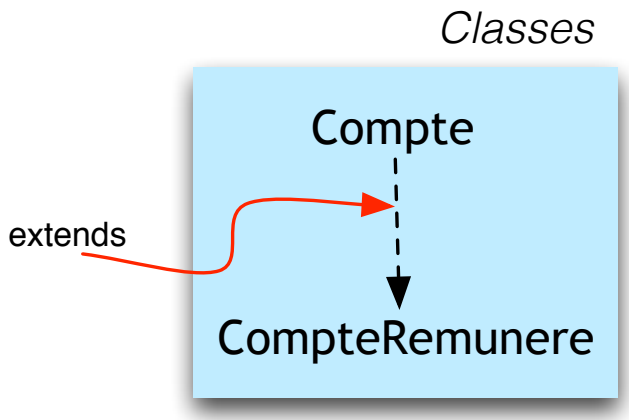
```
class CompteRemunere extends Compte {
  double taux;
  ...
  void fixeTaux(double t) { this.taux = t;}
  void crediterInterets() { ...}
}
```

Un objet `CompteRemunere` contient attributs + méthodes de sa classe *plus* ceux de la classe `Compte`.

`CompteRemunere` hérite de toutes les méthodes et attributs de `Compte`.

Exemple : hiérarchie entre classes (2)

```
class CompteRemunere extends Compte {
  double taux;
  void fixeTaux(double t){…}
  void crediterInterets(){…}
}
```



Exemple de polymorphisme par sous-typage (2)

```
public static void main(...){
   Compte c = new Compte();
   CompteRemunerere cr = new CompteRemunere();
   cr.depot(20);
   c.depot(50);
```

`depot` s'applique aussi bien sur un Compte que sur un CompteRemunere.

`depot` est polymorphe par sous-typage : elle s'applique sur tout objet descendant du type `Compte`.

Exemple de polymorphisme par sous-typage (3)

```
static double somme(Compte c1, Compte c2){
  return (c1.getSolde() + c2.getSolde());
}

public static void main(...){
  Compte c = new Compte();
  CompteRemunerere cr = new CompteRemunere();
  double s1 = somme(c,cr);
  double s2 = somme(cr, cr);
```

`somme` accepte aussi bien des Compte que des CompteRemunere.

`somme` est polymorphe par sous-typage : elle accepte tout objet descendant du type `Compte` déclaré pour ses arguments.

Sous-typage et affectation

`Affectation T x = v;`

Une variable *x* déclarée de type T, peut être affectée par une valeur *v*, si le type Q de *v* est un sous-type de T (identique ou descendant de T).

```
Compte c = new CompteRemunere();
Affichable a = new Compte();
```

- `CompteRemunere` est un sous-type de `Compte` ;
- `Compte` est un sous-type de `Affichable` ;

Sous-typage, collections, tableaux

Supposons :

- *lc* est une collection d'éléments `<T>`
- *t* est un tableau de type `[]T`

On peut affecter dans une case de *t* ou de *lc*, une valeur sous-type de T.

```
ArrayList<Compte> lc = new ...;
ArrayList<Affichable> la = new ...;
Compte [] t = new Compte [4];
Compte c = new Compte();
Date d = new Date();
CompteRemunere cr = new CompteRemunere();
lc.add(cr);  lc.add(c);
la.add(c);   la.add(d);
t[0] = c;  t[1] = cr;
```

Le type `Object`

`Object` : racine de la hiérarchie (arbre) de classes.

```
public class Object {
   /* Teste l'egalite entre objet courant et o*/
   boolean equals (Object o){ ...}
   /* Renvoi representation sous forme de chaine */
   void toString() {...}
   ... // autres methodes
}
```

- définit un ensemble de méthodes utiles ;
- toute classe non introduite par `extends` *étend implicitement* `Object` ;
- donc, toute classe hérite des méthodes dans `Object` ;

Bonne pratique : redefinir les méthodes de Object pour la classe courante.

Hiérarchie de classes avec `Object`