Héritage, polymorphisme, liaison dynamique

Explications et méthode
Christopher Henard

1 Héritage

1.1 Présentation

L'héritage permet de définir une nouvelle classe, dite *classe dérivée*, à partir d'une classe existante dite *classe de base*. Cette nouvelle classe hérite d'emblée des fonctionnalités de la classe de base (attributs et méthodes) qu'elle pourra modifier ou compléter à volonté, sans qu'il soit nécéssaire de remettre en question la classe de base.

La notion d'héritage sert à traduire le fait que certaines classes (les *classes dérivées* ou *classes filles* ou *sous-classes*) **particularisent** des classes (les *classes de base* ou *classes mères* ou *superclasses*). On dit aussi que les classes mères généralisent les classes filles.

Exemple : La classe `Voiture` peut hériter de la classe `Vehicule`. Dans ce cas, la sous-classe est `Voiture` et la superclasse est `Vehicule`. Toute voiture est un véhicule. La classe `Voiture` particularise la classe `Vehicule`.

1.2 Principes

1.2.1 Principes de base :

- Le mot clé `extends` indique que l'on hérite d'une classe : `class Voiture extends Vehicule`.
- Si `B extends A`, alors tout `B` est un `A`.
- Le mot clé `super` indique la classe mère.
- Chaque classe a **une et une seule classe mère** ou superclasse (pas d'héritage multiple) dont elle hérite des attributs et des méthodes.
- Par défaut, Java offre un parent à toutes les classes : la classe `Object`.

1.2.2 La classe qui hérite peut :

- Ajouter des attributs, des méthodes et des constructeurs
- **redéfinir** des méthodes : même nom, même signature(profil) et même type retour
- **surcharger** des méthodes : même nom mais signature différente
- Mais elle ne peut retirer aucune variable ou méthode.

1.2.3 La première instruction d'un constructeur peut être un appel :

- à un constructeur de la classe mère : `super(...)`
- ou à un autre constructeur de la classe : `this(...)`.

Sinon, le compilateur fait un appel implicite au constructeur sans paramètre (`super()`) de la classe mère : erreur s'il nexiste pas !

Un constructeur de la classe mère est toujours exécuté avant les autres instructions du constructeur.
La première instruction dun constructeur de la classe mère est l'appel à un constructeur de la classe « grand-mère », et ainsi de suite... Ainsi la toute, toute première instruction dun constructeur qui est exécutée est le constructeur (sans paramètre) de la classe `Object` : cest le seul qui sait comment créer un nouvel objet en mémoire !

1.2.4 Constructeur par défaut d'une classe :

- Si une classe na pas de constructeur explicite, elle possède un constructeur par défaut dont le code ne comprend aucune instruction
- Ce constructeur par défaut n'appelle pas explicitement un constructeur de la classe mère : un appel du constructeur sans paramètre de la classe mère est automatiquement effectué.

1.2.5 De quoi hérite une classe :

- Si une classe `B` hérite de `A` (`B extends A`), elle hérite automatiquement et implicitement de tous les membres de la classe `A` (mais pas des constructeurs)
- Cependant la classe `B` peut ne pas avoir accès à certains membres dont elle a hérité (par exemple, les membres `private`).

1.2.6 Protection :

- `protected` joue sur l'accessibilité des membres (variables ou méthodes) dont une classe a hérité
- Un membre protected `m` d'une classe mère `A` est accessible par toutes les classes filles de `A` mais est également accessible par toutes les classes du même paquetage que la classe `A`.

2 Polymorphisme et liaison dynamique

2.1 Polymorphisme

Le polymorphisme est le fait qu'une même écriture peut correspondre à différents appels de méthodes.
Exemple :

```
A a = x.f();
a.m();
```

`f` peut renvoyer une instance de `A` ou de nimporte quelle sous-classe de `A`.
`a.m()` peut appeler la méthode `m()` de `A` ou de nimporte quelle sous-classe de `A`.

2.2 Type statique et type dynamique

```
Vehicule v = new Voiture();
```

v est de type statique `Vehicule` et de type dynamique `Voiture`.

2.3 Liaison dynamique

Le polymorphisme est obtenu grâce au « late binding » (**liaison dynamique** ou **retardée**), c'est à dire que la méthode qui sera exécutée est déterminée :

- seulement à l'exécution, et pas dès la compilation.
- par le type réel (type dynamique) de l'objet qui reçoit le message (et pas par son type déclaré (type statique)).

2.4 Type statique et polymorphisme

Le typage statique garantit dès la compilation l'existence de la méthode appelée : la classe déclarée (qui correspond au type statique) de l'objet qui reçoit le message **doit posséder** une méthode du même nom que la méthode appelée.

2.5 Valeurs de retour covariantes (depuis Java 5.0)

Comme vu précédemment, pour qu'il y ait une **redéfinition** de méthode entre une classe `A` et sa fille `B`, il doit y avoir identité du profil donc en particulier du type de valeur de retour. Java 5.0 introduit une exception à cette règle : la nouvelle méthode (la redéfinition) peut renvoyer une valeur d'un type identique **ou dérivé** de celui de la méthode qu'elle redéfinit.

Exemple :

```
class A{
public A f(){...}
...
}

class B{
public B f(){...} //B.f redéfinit bien A.f
...
}
```

`f` appliquée à un objet de type `A` fournit un résultat de type `A`.
`f` appliquée à un objet de type `B` fournit un résultat de type `B`.
Malgré le profil différent au niveau du type de retour, c'est une redéfinition. Avant Java 5, cela aurait été une surcharge.

2.6 Mécanisme de liaison dynamique

Dans un appel de la forme `x.f(...)` ou `x` est supposée de classe `T`, le choix de `f` est déterminé ainsi :

- à la compilation : on détermine **dans la classe** `T` ou ses ascendantes la signature de la **meilleure méthode** `f` convenant à l'appel, ce qui définit du même coup le type de la valeur de retour. Pour les paramètres, on regarde le type statique et leur nombre.
- à l'exécution : on recherche la méthode `f` de signature et de type de retour voulus (avec possibilité de covariance depuis Java 5.0), à partir de la classe correspondant au type dynamique de l'objet référencé par `x` (il est obligatoirement de type `T` ou descendant) ; si cette classe ne comporte pas de méthode appropriée, on remonte dans la hiérarchie jusqu'à ce qu'on en trouve une (au pire, on remonte jusqu'à `T`).

2.7 Application à l'exercice 4 du TD6

```
class A {
   void m(A a){
     System.out.println("m de A");
   }
   void n(A a){
     System.out.println("n de A");
   }
}

class B extends A {
   void m(A a){
     System.out.println("m de B");
   }
   void n(B b){
     System.out.println("n de B");
   }
}

class Test{
   public static void main(String[] argv){
     A a = new B();
     B b = new B();
     A a1 = new A();
     a.m(b);
     a.n(b);
     b.m(b);
```

```
    b.n(b);
    a.m(a1);
    a.n(a1);
    a1.m(b);
    a1.n(b);
  }
}
```

Déterminons les résultats en appliquant le mécanisme de liaison dynamique.

2.7.1 a.m(b)

- À la compilation, la signature est déterminée dans `A` puisque `a` est de type statique `A` : cette signature ou profil est `void m(A a)` car bien qu'on applique m sur un objet de type dynamique `B`, c'est cette signature qui correspond dans la classe `A` (et le type statique de `b` est `A`).
- À l'exécution, vu que `a` est de type dynamique `B`, on regarde dans `B` si on trouve une méthode de profil `void m(A a)`. Oui. C'est donc "m de B"' qui sera affiché.

2.7.2 a.n(b)

- À la compilation, la signature est déterminée dans `A` puisque `a` est de type statique `A` : comme précédemment, cette signature ou profil est `void n(A a)`.
- À l'exécution, vu que `a` est de type dynamique `B`, on regarde dans `B` si on trouve une méthode de profil `void n(A a)`. Non. L'erreur ici serait de croire que `void n(B b)` peut correspondre car on applique n sur un objet de type `B`. On remonte donc dans `A` et c'est "n de A" qui sera affiché.

2.7.3 b.m(b)

- À la compilation, la signature est déterminée dans `B` puisque `b` est de type statique `B` : cette signature ou profil est `void m(A a)`.
- À l'exécution, vu que `B` est même type dynamique que statique, on sait directement que m de `B` sera exécuté : "m de B" sera affiché.

2.7.4 b.n(b)

- À la compilation, la signature est déterminée dans `B` puisque `b` est de type statique `B` : cette signature ou profil est `void n(B b)`.
- À l'exécution, vu que `B` est même type dynamique que statique, on sait directement que n de `B` sera exécuté : "n de B" sera affiché.

2.7.5 a.m(a1)

- À la compilation, la signature est déterminée dans `A` puisque `a` est de type statique `A` : comme précédemment, cette signature ou profil est `void m(A a)`.
- À l'exécution, vu que `a` est de type dynamique `B`, on regarde dans `B` si on trouve une méthode de profil `void m(A a)`. Oui. C'est donc "m de B"' qui sera affiché.

2.7.6 a.n(a1)

- À la compilation, la signature est déterminée dans `A` puisque `a` est de type statique `A` : comme précédemment, cette signature ou profil est `void n(A a)`.
- À l'exécution, vu que `a` est de type dynamique `B`, on regarde dans `B` si on trouve une méthode de profil `void n(A a)`. Non (comme précédemment). On remonte dans `A`. C'est donc "n de A"' qui sera affiché.

2.7.7 a1.m(b)

- À la compilation, la signature est déterminée dans `A` puisque `a1` est de type statique `A` : cette signature ou profil est `void m(A a)`.
- À l'exécution, vu que `a1` est de même type dynamique que statique, on appelle `m` de `A`. C'est donc "m de A"' qui sera affiché.

2.7.8 a1.n(b)

- À la compilation, la signature est déterminée dans `A` puisque `a1` est de type statique `A` : cette signature ou profil est `void n(A a)`.
- À l'exécution, vu que `a1` est de même type dynamique que statique, on appelle `n` de `A`. C'est donc "n de A"' qui sera affiché.

On rajoute une méthode dans A :

```
class A {
   void m(A a){
     System.out.println("m de A");
   }
   void m(B b){
     System.out.println("m de A version 2");
   }
   void n(A a){
     System.out.println("n de A");
   }
}
```

Déterminons les nouveaux résultats en appliquant le mécanisme de liaison dynamique.

2.7.9 a.m(b)

- À la compilation, la signature est déterminée dans `A` puisque `a` est de type statique `A` : cette signature ou profil existant dans `A` est `void m(B b)`.
- À l'exécution, vu que `a` est de type dynamique `B`, on regarde dans `B` si on trouve une méthode de profil `void m(B b)`. Non. On remonte dans `A`. C'est donc "m de A version 2"' qui sera affiché.

2.7.10 a.n(b)

- À la compilation, la signature est déterminée dans `A` puisque `a` est de type statique `A` : comme précédemment, cette signature ou profil est `void n(A a)`.
- À l'exécution, vu que `a` est de type dynamique `B`, on regarde dans `B` si on trouve une méthode de profil `void n(A a)`. Non. L'erreur ici serait de croire que `void n(B b)` peut correspondre car on applique n sur un objet de type `B`. On remonte donc dans `A` et c'est "n de A" qui sera affiché.

2.7.11 b.m(b)

- À la compilation, la signature est déterminée dans `B` puisque `b` est de type statique `B` : cette signature ou profil est `void m(B b)`.
- À l'exécution, vu que `B` est même type dynamique que statique, on sait directement que m de `B` sera exécuté : "m de A version 2" sera affiché car la classe `B` a hérité de la méthode void `m(B b)` de `A` (surcharge).

2.7.12 b.n(b)

- À la compilation, la signature est déterminée dans `B` puisque `b` est de type statique `B` : cette signature ou profil est `void n(B b)`.
- À l'exécution, vu que `B` est même type dynamique que statique, on sait directement que n de `B` sera exécuté : "n de B" sera affiché.

2.7.13 a.m(a1)

- À la compilation, la signature est déterminée dans `A` puisque `a` est de type statique `A` : comme précédemment, cette signature ou profil est `void m(A a)`.
- À l'exécution, vu que `a` est de type dynamique `B`, on regarde dans `B` si on trouve une méthode de profil `void m(A a)`. Oui. C'est donc "m de B"' qui sera affiché.

2.7.14 a.n(a1)

- À la compilation, la signature est déterminée dans `A` puisque `a` est de type statique `A` : comme précédemment, cette signature ou profil est `void n(A a)`.
- À l'exécution, vu que `a` est de type dynamique `B`, on regarde dans `B` si on trouve une méthode de profil `void n(A a)`. Non (comme précédemment). On remonte dans `A`. C'est donc "n de A"' qui sera affiché.

2.7.15 a1.m(b)

- À la compilation, la signature est déterminée dans `A` puisque `a1` est de type statique `A` : cette signature ou profil est `void m(B b)`.
- À l'exécution, vu que `a1` est de même type dynamique que statique, on appelle `m` de `A`. C'est donc "m de A version 2"' qui sera affiché.

2.7.16 a1.n(b)

- À la compilation, la signature est déterminée dans `A` puisque `a1` est de type statique `A` : cette signature ou profil est `void n(A a)`.
- À l'exécution, vu que `a1` est de même type dynamique que statique, on appelle `n` de `A`. C'est donc "n de A"' qui sera affiché.