Programmation objet en Java

Vincent Vidal

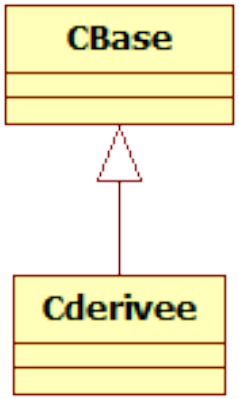
Maître de Conférences
Enseignements : IUT Lyon 1 - pôle AP - Licence ESSIR - bureau 2ème étage
Recherche : Laboratoire LIRIS - bât. Nautibus
E-mail : vincent.vidal@univ-lyon1.fr

Supports de cours et TPs : `http://spiralconnect.univ-lyon1.fr` module "S2 - M2103 - Java et POO"
48H prévues $\approx$ 39H de cours-TDs + 6H de TPs, 1H - interros, et 2H - examen final

Évaluation : Contrôle continu + examen final + Bonus/Malus TP

Plan

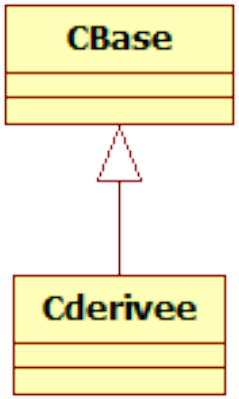
Classes dérivée et de base



En POO il est possible de créer une classe, appelée **classe dérivée**, à partir d'une classe de référence, appelée **classe de base**.

La classe dérivée va hériter des potentialités (**services**=méthodes publiques) de la classe de base.

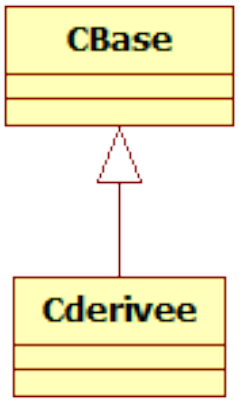
Classes dérivée et de base



En POO il est possible de créer une classe, appelée **classe dérivée**, à partir d'une classe de référence, appelée **classe de base**.

La classe dérivée va hériter des potentialités (**services**=méthodes publiques) de la classe de base.

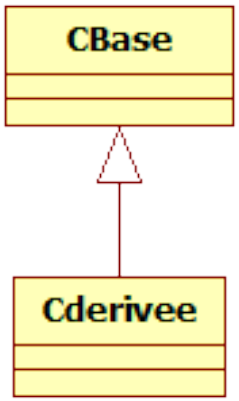
Classes dérivée et de base



La classe dérivée pourra aussi **modifier le comportement des services hérités si nécessaire** et **en rajouter à volonté**.

On va pouvoir créer plusieurs classes filles/dérivées à partir de la même classe mère/de base.

Classes dérivée et de base



La classe dérivée pourra aussi **modifier le comportement des services hérités si nécessaire** et **en rajouter à volonté**.

On va pouvoir créer plusieurs classes filles/dérivées à partir de la même classe mère/de base.

Intérêts de l'héritage ?

L'héritage facilite donc la **réutilisabilité**, **et aussi la maintenance logiciel**, en permettant de créer une nouvelle version d'un logiciel en "héritant" de la version précédente.

L'héritage permet plus que cela : il peut forcer des classes dérivées de classes particulières (des *interfaces*) à implémenter certains services.

Intérêts de l'héritage ?

L'héritage facilite donc la **réutilisabilité**, **et aussi la maintenance logiciel**, en permettant de créer une nouvelle version d'un logiciel en "héritant" de la version précédente.

L'héritage permet plus que cela : il peut forcer des classes dérivées de classes particulières (des *interfaces*) à implémenter certains services.

Héritage VS composition/agrégation

On peut "concevoir" l'héritage en partie comme une composition avec un objet parent "caché" :

les méthodes publiques de cet objet parent, dont hérite le fils, sont appelées directement sur l'objet caché.

Héritage VS composition/agrégation

On peut "concevoir" l'héritage en partie comme une composition avec un objet parent "caché" :
les méthodes publiques de cet objet parent, dont hérite le fils, sont appelées directement sur l'objet caché.

Héritage VS composition/agrégation

L'héritage induit en plus de cette notion d'objet parent caché, une **notion de filiation/compatibilité des types** :

Une classe dérivée est d'un type compatible avec sa classe de base :
une variable de type référence sur un objet de type Base peut "recevoir" une référence sur un type Derivee ;

```
Base o = new Derivee(...) ; // si Derivee fille de Base
```

Chaque méthode nécessitant un argument du type référence sur un objet de base, pourra donc recevoir également une référence sur un type enfant (une sous-classe).
Cette filiation suit une logique temporelle :
la mère doit être créée avant la fille.

Héritage VS composition/agrégation

L'héritage induit en plus de cette notion d'objet parent caché, une **notion de filiation/compatibilité des types** :
Une classe dérivée est d'un type compatible avec sa classe de base :
une variable de type référence sur un objet de type Base peut "recevoir" une référence sur un type Derivee ;
`Base o = new Derivee(...) ; // si Derivee fille de Base`

Chaque méthode nécessitant un argument du type référence sur un objet de base, pourra donc recevoir également une référence sur un type enfant (une sous-classe).
Cette filiation suit une logique temporelle :
la mère doit être créée avant la fille.

Héritage VS composition/agrégation

L'héritage induit en plus de cette notion d'objet parent caché, une **notion de filiation/compatibilité des types** :
Une classe dérivée est d'un type compatible avec sa classe de base :
une variable de type référence sur un objet de type Base peut "recevoir" une référence sur un type Derivee ;

```
Base o = new Derivee(...) ; // si Derivee fille de Base
```

Chaque méthode nécessitant un argument du type référence sur un objet de base, pourra donc recevoir également une référence sur un type enfant (une sous-classe).
Cette filiation suit une logique temporelle :
la mère doit être créée avant la fille.

Héritage VS composition/agrégation

L'héritage induit en plus de cette notion d'objet parent caché, une **notion de filiation/compatibilité des types** :
Une classe dérivée est d'un type compatible avec sa classe de base :
une variable de type référence sur un objet de type Base peut "recevoir" une référence sur un type Derivee ;
Base o = new Derivee(...) ; // si *Derivee* fille de *Base*

Chaque méthode nécessitant un argument du type référence sur un objet de base, pourra donc recevoir également une référence sur un type enfant (une sous-classe).
Cette filiation suit une logique temporelle :
la mère doit être créée avant la fille.

Héritage VS composition/agrégation

L'héritage induit en plus de cette notion d'objet parent caché, une **notion de filiation/compatibilité des types** :
Une classe dérivée est d'un type compatible avec sa classe de base :
une variable de type référence sur un objet de type Base peut "recevoir" une référence sur un type Derivee ;
Base o = new Derivee(...) ; // si *Derivee* fille de *Base*

Chaque méthode nécessitant un argument du type référence sur un objet de base, pourra donc recevoir également une référence sur un type enfant (une sous-classe).
Cette filiation suit une logique temporelle :
la mère doit être créée avant la fille.

Héritage VS composition/agrégation

L'héritage induit en plus de cette notion d'objet parent caché, une **notion de filiation/compatibilité des types** :
Une classe dérivée est d'un type compatible avec sa classe de base :
une variable de type référence sur un objet de type Base peut "recevoir" une référence sur un type Derivee ;
Base o = new Derivee(...) ; // si *Derivee* fille de *Base*

Chaque méthode nécessitant un argument du type référence sur un objet de base, pourra donc recevoir également une référence sur un type enfant (une sous-classe).
Cette filiation suit une logique temporelle :
la mère doit être créée avant la fille.

Exemple simple avec constructeur par défaut

```
class Point2D
{
    public void initialise(int abs, int ord){ m_x = abs ; m_y = ord ; }
    public void deplace(int dx, int dy){ m_x += dx ; m_y += dy ; }
    public void affiche (){ System.out.println( "(" + m_x + "," + m_y + ")" ) ; }

    private int m_x, m_y ; // 2 membres privés
}
```

Cette classe permet de représenter des points/positions sur le plan (2D).

***Nous aimerions avoir une nouvelle classe avec les mêmes fonctionnalités que la classe Point2D, tout en en définissant de nouvelles**, par exemple la gestion d'un attribut couleur.*

Exemple simple avec constructeur par défaut

```
class Point2D
{
    public void initialise(int abs, int ord){ m_x = abs ; m_y = ord ; }
    public void deplace(int dx, int dy){ m_x += dx ; m_y += dy ; }
    public void affiche (){ System.out.println( "(" + m_x + "," + m_y + ")" ) ; }

    private int m_x, m_y ; // 2 membres privés
}
```

Cette classe permet de représenter des points/positions sur le plan (2D).

***Nous aimerions avoir une nouvelle classe avec les mêmes fonctionnalités que la classe Point2D, tout en en définissant de nouvelles**, par exemple la gestion d'un attribut couleur*.

Exemple simple avec constructeur par défaut

```
class Point2DCoul extends Point2D
{
    public void colore(byte couleur) { m_couleur = couleur ; }
    private byte m_couleur ; // code représentant une couleur
}
```

La déclaration class Point2DCoul extends Point2D spécifie que Point2DCoul est une classe dérivée de la classe de base Point2D.

Les membres (attributs et méthodes) hérités depuis la classe de base Point2D conserveront la même accessibilité (public, private) au sein de la classe dérivée Point2DCoul.

Exemple simple avec constructeur par défaut

```
class Point2DCoul extends Point2D
{
    public void colore(byte couleur) { m_couleur = couleur ; }
    private byte m_couleur ; // code représentant une couleur
}
```

La déclaration class Point2DCoul extends Point2D spécifie que Point2DCoul est une classe dérivée de la classe de base Point2D.

Les membres (attributs et méthodes) hérités depuis la classe de base Point2D conserveront la même accessibilité (public, private) au sein de la classe dérivée Point2DCoul.

Exemple simple avec constructeur par défaut

```
class Point2DCoul extends Point2D
{
    public void colore(byte couleur) { m_couleur = couleur ; }
    private byte m_couleur ; // code représentant une couleur
}
```

La déclaration class Point2DCoul extends Point2D spécifie que Point2DCoul est une classe dérivée de la classe de base Point2D.

Les membres (attributs et méthodes) hérités depuis la classe de base Point2D conserveront la même accessibilité (public, private) au sein de la classe dérivée Point2DCoul.

Exemple simple avec constructeur par défaut

```
...
    Point2DCoul pcol = new Point2DCoul() ;
    pcol.affiche() ; // OK
    pcol.initialise(2, -5) ; // OK
    pcol.colore((byte)47) ; // OK
    pcol.deplace(2, -4) ; // OK
    pcol.affiche() ; // OK
...
```

pcol a accès :

- aux méthodes publiques de la classe Point2DCoul ;
- aux méthodes publiques de la classe Point2D.

Exemple simple avec constructeur par défaut

```
...
    Point2DCoul pcol = new Point2DCoul() ;
    pcol.affiche() ; // OK
    pcol.initialise(2, -5) ; // OK
    pcol.colore((byte)47) ; // OK
    pcol.deplace(2, -4) ; // OK
    pcol.affiche() ; // OK
...
```

pcol a accès :

- aux méthodes publiques de la classe Point2DCoul ;
- aux méthodes publiques de la classe Point2D.

Exemple simple avec constructeur par défaut

```
...
    Point2DCoul pcol = new Point2DCoul() ;
    pcol.affiche() ; // OK
    pcol.initialise(2, -5) ; // OK
    pcol.colore((byte)47) ; // OK
    pcol.deplace(2, -4) ; // OK
    pcol.affiche() ; // OK
...
```

pcol a accès :

- aux méthodes publiques de la classe Point2DCoul ;
- aux méthodes publiques de la classe Point2D.

Exemple simple avec constructeur par défaut

```
...
    Point2DCoul pcol = new Point2DCoul() ;
    pcol.affiche() ; // OK
    pcol.initialise(2, -5) ; // OK
    pcol.colore((byte)47) ; // OK
    pcol.deplace(2, -4) ; // OK
    pcol.affiche() ; // OK
...
```

pcol a accès :

- aux méthodes publiques de la classe Point2DCoul ;
- aux méthodes publiques de la classe Point2D.

Remarques

- La classe dérivée peut être la classe de base d'une autre classe.
- La classe de base est également appelée *super-classe*.
- La classe de base doit être entièrement connue au moment de la dérivation :
  soit la classe de base est dans le même fichier, soit elle est déjà compilée avec son bytecode accessible.

Remarques

- La classe dérivée peut être la classe de base d'une autre classe.
- La classe de base est également appelée *super-classe*.
- La classe de base doit être entièrement connue au moment de la dérivation :
  soit la classe de base est dans le même fichier, soit elle est déjà compilée avec son bytecode accessible.

Remarques

- La classe dérivée peut être la classe de base d'une autre classe.
- La classe de base est également appelée *super-classe*.
- La classe de base doit être entièrement connue au moment de la dérivation :
  soit la classe de base est dans le même fichier, soit elle est déjà compilée avec son bytecode accessible.

Remarques

- La classe dérivée peut être la classe de base d'une autre classe.
- La classe de base est également appelée *super-classe*.
- La classe de base doit être entièrement connue au moment de la dérivation :
  soit la classe de base est dans le même fichier, soit elle est déjà compilée avec son bytecode accessible.

Respect de l'encapsulation

Une méthode d'une classe dérivée :

- **peut accéder** aux membres public de sa classe de base (membres=attributs+méthodes).
- **ne peut pas accéder** aux membres private de sa classe de base.

Si on munit la classe Point2DCoul d'une méthode afficheC :

```
public void afficheC(){
   System.out.println( "(" + m_x + "," + m_y + "," + m_couleur + ")" ) ;   // ERREUR car m_x et
}                                                                          // m_y private
```

On doit utiliser la méthode affiche de la classe Point2D...

Respect de l'encapsulation

Une méthode d'une classe dérivée :

- **peut accéder** aux membres public de sa classe de base (membres=attributs+méthodes).
- **ne peut pas accéder** aux membres private de sa classe de base.

Si on munit la classe Point2DCoul d'une méthode afficheC :

```
public void afficheC(){
   System.out.println( "(" + m_x + "," + m_y + "," + m_couleur + ")" ) ;   // ERREUR car m_x et
}                                                                          // m_y private
```

On doit utiliser la méthode affiche de la classe Point2D...

Respect de l'encapsulation

Une méthode d'une classe dérivée :

- **peut accéder** aux membres public de sa classe de base (membres=attributs+méthodes).
- **ne peut pas accéder** aux membres private de sa classe de base.

Si on munit la classe Point2DCoul d'une méthode afficheC :

```
public void afficheC(){
   System.out.println( "(" + m_x + "," + m_y + "," + m_couleur + ")" ) ;    // ERREUR car m_x et
}                                                                            // m_y private
```

On doit utiliser la méthode affiche de la classe Point2D...

Respect de l'encapsulation

Une méthode d'une classe dérivée :

- **peut accéder** aux membres public de sa classe de base (membres=attributs+méthodes).
- **ne peut pas accéder** aux membres private de sa classe de base.

Si on munit la classe Point2DCoul d'une méthode afficheC :

```
public void afficheC(){
   System.out.println( "(" + m_x + "," + m_y + "," + m_couleur + ")" ) ;    // ERREUR car m_x et
}                                                                            // m_y private
```

On doit utiliser la méthode affiche de la classe Point2D...

Respect de l'encapsulation

La méthode afficheC de la classe Point2DCoul doit utiliser la méthode affiche de la classe Point2D :

```
public void afficheC(){
   affiche() ;            // équivalent à this.affiche() ;
   System.out.println( "couleur = " + m_couleur ) ;
}
```

```
...
public void initialiseC(int abs, int ord, byte coul){
   initialise(abs, ord) ; // équivalent à this.initialise(abs, ord) ;
   m_couleur = coul ;
}
```

Respect de l'encapsulation

La méthode afficheC de la classe Point2DCoul doit utiliser la méthode affiche de la classe Point2D :

```
public void afficheC(){
   affiche() ;            // équivalent à this.affiche() ;
   System.out.println( "couleur = " + m_couleur ) ;
}
```

```
...
public void initialiseC(int abs, int ord, byte coul){
   initialise(abs, ord) ; // équivalent à this.initialise(abs, ord) ;
   m_couleur = coul ;
}
```

Différentes situations possibles

- La classe de base dispose ou pas d'un constructeur différent du constructeur par défaut.
- La classe dérivée dispose ou pas d'un constructeur différent du constructeur par défaut.

Différentes situations possibles

- La classe de base dispose ou pas d'un constructeur différent du constructeur par défaut.
- La classe dérivée dispose ou pas d'un constructeur différent du constructeur par défaut.

Cas n°1 : les classes de base et dérivée disposent d'un contructeur différent du constructeur par défaut

Règle : Le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet :
il appellera le constructeur de la classe de base avec les paramètres adéquates, et il complètera éventuellement avec l'appel à des setters pour les champs encapsulés de la classe de base.

Règle : Si le constructeur de la classe dérivée utilise/appelle un constructeur de la classe de base, alors cet appel doit être obligatoirement la première instruction et doit se faire via le mot-clé *super*.

Cas n°1 : les classes de base et dérivée disposent d'un contructeur différent du constructeur par défaut

Règle : Le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet :
il appellera le constructeur de la classe de base avec les paramètres adéquates, et il complètera éventuellement avec l'appel à des setters pour les champs encapsulés de la classe de base.

Règle : Si le constructeur de la classe dérivée utilise/appelle un constructeur de la classe de base, alors cet appel doit être obligatoirement la première instruction et doit se faire via le mot-clé *super*.

Cas n°1 : les classes de base et dérivée disposent d'un contructeur différent du constructeur par défaut

Règle : Le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet :
il appellera le constructeur de la classe de base avec les paramètres adéquates, et il complètera éventuellement avec l'appel à des setters pour les champs encapsulés de la classe de base.

Règle : Si le constructeur de la classe dérivée utilise/appelle un constructeur de la classe de base, alors cet appel doit être obligatoirement la première instruction et doit se faire via le mot-clé *super*.

Cas n°1 : les classes de base et dérivée disposent d'un contructeur différent du constructeur par défaut

```
class Point2D
{
    public Point2D(int abs, int ord)
    {  ... // partie spécifique à Point2D
    }
    ...
    private int m_x, m_y ; // coordonnées
}

class Point2DCoul extends Point2D
{
    public Point2DCoul(int abs, int ord, byte coul)
    {
       super(abs, ord) ; // appel constructeur classe de base obligatoirement en 1ère instruction
       ... // partie spécifique à Point2DCoul
    }
    ...
    private byte m_couleur ; // code représentant une couleur
}
```

Cas n°1 : les classes de base et dérivée disposent d'un contructeur différent du constructeur par défaut

```
public class TestPoint2DCoul{
   public static void main(String args[])
   {
      Point2DCoul pcol = new Point2DCoul(2, -5, (byte)47) ;
      pcol.afficheC() ; // OK
      pcol.deplace(2, -4) ; // OK
      pcol.afficheC() ; // OK
   }
}
```

Grâce aux constructeurs, nous n'avons plus besoin des méthodes initialise et initialiseC.

Cas n°1 : les classes de base et dérivée disposent d'un contructeur différent du constructeur par défaut

```
public class TestPoint2DCoul{
   public static void main(String args[])
   {
      Point2DCoul pcol = new Point2DCoul(2, -5, (byte)47) ;
      pcol.afficheC() ; // OK
      pcol.deplace(2, -4) ; // OK
      pcol.afficheC() ; // OK
   }
}
```

Grâce aux constructeurs, nous n'avons plus besoin des méthodes initialise et initialiseC.

Cas n°2 : la classe de base possède uniquement le constructeur par défaut

- Il reste possible de mettre un $\mathrm{super}();$ en 1ère instruction d'un constructeur de la classe dérivée.
  Cela forcera à maintenir l'existence d'un constructeur sans argument dans la classe de base et donc forcera un développeur ajoutant un constructeur à la classe de base d'ajouter aussi un constructeur adéquate sans argument.
- Si la classe dérivée possède uniquement le constructeur par défaut, alors l'appel au constructeur par défaut de la classe de base est automatique. *Ces appels ne feront rien.*

Cas n°2 : la classe de base possède uniquement le constructeur par défaut

- Il reste possible de mettre un `super();` en 1ère instruction d'un constructeur de la classe dérivée.
  Cela forcera à maintenir l'existence d'un constructeur sans argument dans la classe de base et donc forcera un développeur ajoutant un constructeur à la classe de base d'ajouter aussi un constructeur adéquate sans argument.
- Si la classe dérivée possède uniquement le constructeur par défaut, alors l'appel au constructeur par défaut de la classe de base est automatique. *Ces appels ne feront rien.*

Cas n°2 : la classe de base possède uniquement le constructeur par défaut

- Il reste possible de mettre un `super();` en 1ère instruction d'un constructeur de la classe dérivée.
  Cela forcera à maintenir l'existence d'un constructeur sans argument dans la classe de base et donc forcera un développeur ajoutant un constructeur à la classe de base d'ajouter aussi un constructeur adéquate sans argument.
- Si la classe dérivée possède uniquement le constructeur par défaut, alors l'appel au constructeur par défaut de la classe de base est automatique. *Ces appels ne feront rien.*

Cas n°2 : la classe de base possède uniquement le constructeur par défaut

- Il reste possible de mettre un `super();` en 1ère instruction d'un constructeur de la classe dérivée.
  Cela forcera à maintenir l'existence d'un constructeur sans argument dans la classe de base et donc forcera un développeur ajoutant un constructeur à la classe de base d'ajouter aussi un constructeur adéquate sans argument.
- Si la classe dérivée possède uniquement le constructeur par défaut, alors l'appel au constructeur par défaut de la classe de base est automatique. *Ces appels ne feront rien*.

Cas n°3 : la classe dérivée possède uniquement le constructeur par défaut

Il y aura un appel automatique au constructeur sans argument de la classe de base :

- soit aucun constructeur n'est défini dans la classe de base auquel cas le constructeur par défaut ne faisant rien est appelé ;
- soit un constructeur public sans argument est défini dans la classe de base et est appelé ;
- soit le programme ne compile pas.

Cas n°3 : la classe dérivée possède uniquement le constructeur par défaut

Il y aura un appel automatique au constructeur sans argument de la classe de base :

- soit aucun constructeur n'est défini dans la classe de base auquel cas le constructeur par défaut ne faisant rien est appelé ;
- soit un constructeur public sans argument est défini dans la classe de base et est appelé ;
- soit le programme ne compile pas.

Cas n°3 : la classe dérivée possède uniquement le constructeur par défaut

Il y aura un appel automatique au constructeur sans argument de la classe de base :

- soit aucun constructeur n'est défini dans la classe de base auquel cas le constructeur par défaut ne faisant rien est appelé ;
- soit un constructeur public sans argument est défini dans la classe de base et est appelé ;
- soit le programme ne compile pas.

Ordre d'initialisation des champs

1. Initialisation par défaut de tous les champs (ceux propres à la classe de base et ceux propres à la classe dérivée).
2. Initialisation explicite des champs hérités de la classe de base.
3. Exécution d'un constructeur de la classe de base.
4. Initialisation explicite des champs propres à la classe dérivée.
5. Exécution d'un constructeur de la classe dérivée.

Ordre d'initialisation des champs

1. Initialisation par défaut de tous les champs (ceux propres à la classe de base et ceux propres à la classe dérivée).
2. Initialisation explicite des champs hérités de la classe de base.
3. Exécution d'un constructeur de la classe de base.
4. Initialisation explicite des champs propres à la classe dérivée.
5. Exécution d'un constructeur de la classe dérivée.

Ordre d'initialisation des champs

1. Initialisation par défaut de tous les champs (ceux propres à la classe de base et ceux propres à la classe dérivée).
2. Initialisation explicite des champs hérités de la classe de base.
3. Exécution d'un constructeur de la classe de base.
4. Initialisation explicite des champs propres à la classe dérivée.
5. Exécution d'un constructeur de la classe dérivée.

Ordre d'initialisation des champs

1. Initialisation par défaut de tous les champs (ceux propres à la classe de base et ceux propres à la classe dérivée).
2. Initialisation explicite des champs hérités de la classe de base.
3. Exécution d'un constructeur de la classe de base.
4. Initialisation explicite des champs propres à la classe dérivée.
5. Exécution d'un constructeur de la classe dérivée.

Ordre d'initialisation des champs

1. Initialisation par défaut de tous les champs (ceux propres à la classe de base et ceux propres à la classe dérivée).
2. Initialisation explicite des champs hérités de la classe de base.
3. Exécution d'un constructeur de la classe de base.
4. Initialisation explicite des champs propres à la classe dérivée.
5. Exécution d'un constructeur de la classe dérivée.

On peut surdéfinir une méthode existante dans une classe de base au sein d'une classe dérivée

La liste des méthodes (avec un même identifiant) surdéfinies "accessibles" pour un objet est la liste des méthodes définies dans sa classe et ses classes ascendantes (ancêtres).

En aucun cas une surdéfinition d'une classe descendante ne sera accessible.

On peut surdéfinir une méthode existante dans une classe de base au sein d'une classe dérivée

La liste des méthodes (avec un même identifiant) surdéfinies "accessibles" pour un objet est la liste des méthodes définies dans sa classe et ses classes ascendantes (ancêtres).

En aucun cas une surdéfinition d'une classe descendante ne sera accessible.

Différence entre la surdéfinition et la redéfinition de méthodes

La *surdéfinition* consiste à proposer une nouvelle version d'une méthode avec une signature différente (cumul de versions différentes), tandis que la *redéfinition* consiste à proposer une nouvelle définition d'une méthode existante dans une classe parent au sein d'une classe fille (substitution d'un ancien comportement à un nouveau).

Une redéfinition impose donc non seulement le même identifiant, mais aussi la même liste de types d'argument formel et le même type de retour.
Depuis Java 5, les valeurs de retour peuvent être covariantes (type dérivé du type original) au lieu d'être identiques.

Différence entre la surdéfinition et la redéfinition de méthodes

La *surdéfinition* consiste à proposer une nouvelle version d'une méthode avec une signature différente (cumul de versions différentes), tandis que la *redéfinition* consiste à proposer une nouvelle définition d'une méthode existante dans une classe parent au sein d'une classe fille (substitution d'un ancien comportement à un nouveau).

Une redéfinition impose donc non seulement le même identifiant, mais aussi la même liste de types d'argument formel et le même type de retour.

Depuis Java 5, les valeurs de retour peuvent être covariantes (type dérivé du type original) au lieu d'être identiques.

Différence entre la surdéfinition et la redéfinition de méthodes

La *surdéfinition* consiste à proposer une nouvelle version d'une méthode avec une signature différente (cumul de versions différentes), tandis que la *redéfinition* consiste à proposer une nouvelle définition d'une méthode existante dans une classe parent au sein d'une classe fille (substitution d'un ancien comportement à un nouveau).

Une redéfinition impose donc non seulement le même identifiant, mais aussi la même liste de types d'argument formel et le même type de retour.
Depuis Java 5, les valeurs de retour peuvent être covariantes (type dérivé du type original) au lieu d'être identiques.

Exemple de redéfinition

On peut redéfinir la méthode affiche de Point2D au sein de Point2DCoul :

```
class Point2D
{
   ...
   public void affiche (){ System.out.println( "(" + m_x + "," + m_y + ")" ) ; }
   ...
   private int m_x, m_y ; // 2 membres privés
}
class Point2DCoul extends Point2D
{
   ...
   public void affiche (){
      affiche() ; // Erreur: définition récursive "infinie"
      System.out.println( "couleur = " + m_couleur ) ;
   }
   ...
   private byte m_couleur ; // code représentant une couleur
}
```

Exemple de redéfinition

On peut redéfinir la méthode affiche de Point2D au sein de Point2DCoul :

```
class Point2D
{
   ...
   public void affiche (){ System.out.println( "(" + m_x + "," + m_y + ")" ) ; }
   ...
   private int m_x, m_y ; // 2 membres privés
}
class Point2DCoul extends Point2D
{
   ...
   public void affiche (){
      super.affiche() ; // Ok; ici super n'est pas obligatoirement en 1ère position
      System.out.println( "couleur = " + m_couleur ) ;
   }
   ...
   private byte m_couleur ; // code représentant une couleur
}
```

Conséquences de la redéfinition de méthodes

Une méthode redéfinie au sein d'une classe dérivée va *masquer* la version de la classe de base dans la classe dérivée, mais aussi dans les classes descendantes de la classe dérivée jusqu'à une éventuelle autre redéfinition.

```
class Point2D{
   ...
   public void affiche (){ System.out.println( "(" + m_x + "," + m_y + ")" ) ; }
   ... }
class Point2DCoul extends Point2D{
   ...
   public void affiche (){
      super.affiche() ; // Ok; ici super n'est pas obligatoirement en 1ère position
      System.out.println( "couleur = " + m_couleur ) ;
   }
   ... }
public class TestPoint2DCoul{
   public static void main(String args[]){
      Point2DCoul p = new Point2DCoul(1, 2, (byte)3) ;
      p.affiche() ; // on appelle forcément la méthode de la classe Point2DCoul
   }
}
```

La redéfinition de méthodes ne peut pas diminuer les droits d'accès

Une méthode publique dans la classe de base ne peut pas être redéfinie privée dans une classe dérivée (*erreur de compilation*).

Le contraire est par contre autorisé. Cela ne remet pas en cause le principe de l'encapsulation des données car la méthode redéfinie n'a pas accès aux membres privés de la classe de base.

La redéfinition de méthodes ne peut pas diminuer les droits d'accès

Une méthode publique dans la classe de base ne peut pas être redéfinie privée dans une classe dérivée (*erreur de compilation*).

Le contraire est par contre autorisé. Cela ne remet pas en cause le principe de l'encapsulation des données car la méthode redéfinie n'a pas accès aux membres privés de la classe de base.

La redéfinition de méthodes ne peut pas diminuer les droits d'accès

Une méthode publique dans la classe de base ne peut pas être redéfinie privée dans une classe dérivée (*erreur de compilation*).

Le contraire est par contre autorisé. Cela ne remet pas en cause le principe de l'encapsulation des données car la méthode redéfinie n'a pas accès aux membres privés de la classe de base.

La redéfinition de méthodes est limitée aux méthodes d'instance

Une méthode de classe (static) ne peut pas être redéfinie (*erreur de compilation*).

Cela a du sens puisque la jvm se sert du type de l'objet appelant pour déterminer la bonne méthode à appeler et que pour une méthode statique il peut ne pas y avoir d'objet appelant.

La redéfinition de méthodes est limitée aux méthodes d'instance

Une méthode de classe (static) ne peut pas être redéfinie (*erreur de compilation*).

Cela a du sens puisque la jvm se sert du type de l'objet appelant pour déterminer la bonne méthode à appeler et que pour une méthode statique il peut ne pas y avoir d'objet appelant.

On ne redéfinit pas des attributs, mais on les duplique

Il est possible d'avoir des attributs de même nom dans les classes dérivée et de base. Chaque attribut d'instance sera bien présent en mémoire.

```
class A{
   public int n ;
}
class B extends A{
   public float n ;
   public void f ()
   {
      n = 1.2f ;
      super.n = 7 ;
   }
}
public class TestAB{
   public static void main(String args[]){
      A a = new A() ;
      B b = new B() ;
      b.f() ;
      b.n = 1.3f ; // forcément le champ de B
      a.n = 5 ;    // forcément le champ de A
   }
}
```