

Réutilisation et Héritage en Java

Emmanuel BRUNO

Université de Toulon, LIS UMR CNRS 7020

emmanuel.bruno@univ-tln.fr

2026-10-14

Table des matières

1 Pourquoi réutiliser ?	1
2 UML et Java	1
3 Association UML	1
4 L'aggrégation en Java	2
5 La délégation	2
6 Exemple de délégation	2
7 Implémentation de la Navigabilité UML en Java	2
8 L'aggrégation et la composition en Java ..	3
9 L'héritage en Java	3
10 Condition d'utilisation	4
11 Conséquences	5
12 Constructeur	5
13 Limitations	6
14 Le polymorphisme	6
15 Le transtypage	6
16 Les classes abstraites	7
17 Les classes abstraites	7
18 Les interfaces	8
19 Les interfaces	8

1 Pourquoi réutiliser ?

L'utilisation du paradigme objet permet une structuration efficace des applications, facilitant ainsi la réutilisation et le partage des composants au sein des projets de conception et de développement. Ce paradigme favorise également l'extension des fonctionnalités existantes, accélérant le développement et simplifiant la maintenance des logiciels. Cependant,

cette approche requiert une planification et une conception initiales plus poussées pour tirer pleinement parti de ses avantages.

2 UML et Java

Dans cette partie, nous nous concentrerons sur les modèles UML structurels qui décrivent les aspects statiques des objets. Nous aborderons principalement deux types de diagrammes : les diagrammes de classes et les diagrammes d'instances. Les diagrammes de classes illustrent les relations et la structure entre différentes classes, tandis que les diagrammes d'instances montrent les objets concrets et leurs relations à un moment donné. Parallèlement, nous examinerons comment ces concepts peuvent être implémentés en Java, en offrant une compréhension pratique de la transition des modèles théoriques UML vers le code concret.

3 Association UML

En analyse, un attribut ne peut pas être une classe. L'association, en revanche, définit une relation entre deux classes, caractérisée par un nom, une cardinalité et éventuellement des rôles. En Java, une association est souvent implémentée en utilisant des références entre objets dans les classes concernées. Cela permet de modéliser les relations entre instances, offrant ainsi une représentation fidèle des interactions

et des dépendances entre les différentes entités du système.

4 L'aggrégation en Java

En Java, les associations entre objets sont représentées à l'aide d'attributs d'instance. Lorsque la cardinalité est déterminée, par exemple n..m, un tableau peut être utilisé pour modéliser la relation. Cependant, pour une flexibilité accrue, il est généralement recommandé d'utiliser des Collections. De plus, les contraintes exprimées dans le modèle UML, telles que l'ordre, l'exclusivité ou les sous-ensembles, doivent être rigoureusement vérifiées par le programme pour garantir la cohérence des relations entre les instances.

```
new Calculateur();
```

```
1 public class Comptable {
2     private Calculateur
   calculateur = new Calculateur();
3
4     public void
   calculerBilan(int... depenses) {
5         System.out.println(
6             String.format("Dépenses:
   %d", calculateur.add(depenses))
7         );
8     }
9 }
```

```
new
Comptable().calculerBilan(12,23,45);
```

5 La délégation

La délégation est une solution efficace de réutilisation en programmation orientée objet. Elle permet à une classe ou une instance d'appeler une méthode d'une autre classe ou d'instancier un objet d'une autre classe pour lui déléguer des traitements spécifiques. Cela signifie qu'une classe (C1) peut utiliser les fonctionnalités d'une autre classe (C2) sans hériter de ses méthodes ou attributs, offrant ainsi une flexibilité accrue dans la conception du code et la réutilisation des fonctionnalités.

6 Exemple de délégation

```
public class Calculateur {
    public int add(final int... valeurs)
    {
        int somme=0;
        for (int valeur:valeurs)
            somme+=valeur;
        return somme;
    }
}
```

7 Implémentation de la Navigabilité UML en Java

La navigabilité en Java se matérialise principalement par l'utilisation d'attributs et d'accessieurs. Les relations entre objets sont représentées par des attributs dans les classes concernées, avec des méthodes d'accès (getters et setters) pour obtenir et modifier les valeurs de ces attributs.

Il est crucial d'être prudent avec les associations bidirectionnelles. Bien que parfois nécessaires, elles ajoutent une couche de complexité supplémentaire et peuvent rendre la maintenance du code plus difficile. Les associations bidirectionnelles doivent être soigneusement gérées pour éviter les incohérences et les problèmes de synchronisation des données.

Bien que les associations bidirectionnelles puissent être utiles, leur utilisation doit être

limitée et bien gérée pour éviter d'ajouter une complexité excessive au système. Il est souvent préférable de simplifier les relations entre objets et d'utiliser des associations unidirectionnelles lorsque cela est possible pour maintenir un code propre et facile à maintenir.

8 L'aggrégation et la composition en Java

L'aggrégation et la composition en programmation orientée objet présentent des distinctions importantes. L'aggrégation permet à un objet de contenir d'autres objets qui peuvent également exister indépendamment. Par exemple, une classe « Équipe » peut agréger plusieurs objets « Membre », où chaque membre peut appartenir à plusieurs équipes. En revanche, la composition est une relation plus forte où un objet est composé d'autres objets qui n'existent que dans le contexte de cet objet composite. Par exemple, une classe « Voiture » est composée d'objets comme « Moteur », « Roues » et « Carrosserie », et leur durée de vie est liée à celle de la voiture elle-même. Techniquement, en Java on utilise le garbage collector pour détruire les objets non référencés, mais il est crucial de gérer soigneusement les relations pour éviter les références inutiles.

9 L'héritage en Java

- On indique que la classe fille étend la classe mère : `extends`

```
public class Velo {
    protected final String nom;
    public Velo(String nom) {this.nom
= nom;}
    public void avancer() {
        System.out.println(
                                String.format("%s
avance. ", nom));
    }
}
```

```
}
}
```

```
public class VeloElectrique extends
Velo {
    /* Augmentation de l'état interne */
    private boolean etatDuMoteur =
false;
    public VeloElectrique(String nom)
{super(nom);};
    public void avancer() {
        System.out.println(
            String.format("%s avance %s
assistance.",
nom, etatDuMoteur?"avec":"sans"));
    }
    /* Nouveau comportement */
    public void inverserMoteur()
{etatDuMoteur=!etatDuMoteur;}
}
```

```
Velo velo1 = new Velo("V1");
velo1.avancer();

VeloElectrique velo2 = new
VeloElectrique("V2");
velo2.inverserMoteur();
velo2.avancer();

Velo velo3 = new
VeloElectrique("V3");
//velo3.inverserMoteur(); //
Expliquer
velo3.avancer();
```

En Java, la gestion de l'héritage est fondamentale pour organiser et structurer les classes de manière hiérarchique. Lorsqu'une classe hérite d'une autre classe, elle acquiert toutes les va-

riables et méthodes publiques et protégées de cette classe parente. Si aucune classe parente n'est spécifiée avec `extends`, la classe fille hérite automatiquement de la classe `Object`, qui est la racine de toutes les classes Java.

Contrairement à certains langages de programmation qui permettent l'héritage multiple, Java restreint cette possibilité pour éviter les ambiguïtés et les conflits potentiels entre les méthodes et attributs hérités de différentes classes parentes. Ainsi, une classe Java ne peut hériter que d'une seule classe parente à la fois, garantissant ainsi une structure hiérarchique claire et évitant les complications liées à l'héritage multiple.

Dans une classe fille en Java, il est possible d'enrichir l'héritage en ajoutant de nouvelles variables d'instance, méthodes, et même des constructeurs supplémentaires. Cela permet à la classe fille d'étendre les fonctionnalités de sa classe parente tout en maintenant la cohérence et la relation sémantique entre les deux. De plus, Java permet à une classe fille de redéfinir les méthodes héritées de sa classe parente. Ce processus, appelé redéfinition de méthode, permet à la classe fille de fournir une implémentation spécifique d'une méthode héritée pour adapter son comportement à des besoins particuliers, tout en respectant la signature de la méthode définie dans la classe parente.

Définition 1. *La redéfinition d'une méthode consiste à définir une méthode ayant la même signature qu'une méthode définie dans une classe ancêtre.*

```
public class A {  
    public void m()  
{System.out.println("je suis un  
A.");}  
}
```

```
public class B extends A {  
    public void m()  
{System.out.println("je suis un  
B.");}  
}
```

```
public class C extends A {  
    public void m(String message)  
{System.out.println("je suis un C.  
"+message);}  
}
```

```
A a1 = new A();  
A a2 = new B();  
A a3 = new C();  
a1.m();  
a2.m();
```

```
a3.m("Hello");
```

10 Condition d'utilisation

En Java, le principe fondamental de l'héritage suit le concept de «est un» (is a), où une classe fille hérite non seulement des attributs et méthodes de sa classe parente, mais représente également une instance de cette classe parente. Par exemple, si une classe `B` étend une classe `A` avec `B extends A`, alors toute instance de `B`, comme `b`, est considérée comme une instance de `A`. Cette relation d'héritage reflète la nature «est un» dans laquelle une classe spécialisée (`B`) peut être vue comme un cas particulier de la classe plus générale (`A`).

En Java, il est crucial de ne pas utiliser l'héritage en dehors de ce contexte spécifique de spécialisation «est un». L'héritage est conçu pour établir une relation de type «est un», ce qui signifie que la classe fille

partage une relation sémantique forte avec sa classe parente. Utiliser l'héritage dans d'autres contextes pourrait conduire à une mauvaise conception et à une structure de classe incohérente, ce qui peut rendre le code plus difficile à maintenir et à comprendre. Ainsi, Java encourage l'utilisation de l'héritage de manière appropriée pour maintenir une hiérarchie claire et cohérente entre les classes.

11 Conséquences

En Java, lorsqu'une classe fille hérite d'une classe mère à l'aide du mot-clé `extends`, elle acquiert tous les membres de la classe mère, y compris les attributs et les méthodes. L'accès à ces membres peut être restreint en fonction des niveaux de protection (`public`, `private`, `protected`) définis dans la classe mère.

Contrairement aux attributs et aux méthodes, les constructeurs ne sont pas hérités. Cela est dû au fait que l'héritage d'un constructeur n'aurait pas de sens, car les constructeurs sont des méthodes spéciales utilisées pour initialiser les objets d'une classe particulière. Néanmoins, dans une classe fille, il est possible d'appeler explicitement un constructeur de sa classe mère à l'aide de `super()`. Cette instruction `super()` doit être la première instruction du constructeur fille si elle est utilisée.

De plus, Java permet d'appeler un autre constructeur de la même classe à l'aide de `this()`. Cette flexibilité permet à une classe d'avoir plusieurs constructeurs qui peuvent être utilisés en fonction des besoins spécifiques sans répéter le code d'initialisation commun.

Il est essentiel de noter que les instructions `super()` et `this()` ne peuvent être que la première instruction d'un constructeur. Cela garantit que l'initialisation de l'objet se fait correctement et dans l'ordre approprié, assu-

rant ainsi la cohérence et la stabilité de l'objet instancié.

12 Constructeur

En Java, lorsqu'aucun `this()` ou `super()` n'est explicitement spécifié dans un constructeur, le compilateur ajoute automatiquement `super()` comme première instruction. Cela signifie que le premier constructeur appelé est toujours celui de la super classe directe.

Ce mécanisme est essentiel pour garantir que chaque instance d'une classe dérivée est correctement initialisée en appelant d'abord le constructeur de sa super classe. Cela assure une initialisation cohérente des objets tout au long de la hiérarchie d'héritage, en suivant la chaîne des constructeurs jusqu'à la classe la plus basique, qui est `Object` si aucune autre super classe n'est spécifiée explicitement.

Ainsi, même si aucun `extends` n'est spécifié dans une classe, elle hérite implicitement de `Object`, et le constructeur par défaut de `Object` est appelé grâce à `super()`. Ce comportement garantit que tous les membres hérités sont initialisés correctement avant que les membres spécifiques à la classe fille ne soient initialisés, assurant ainsi la cohérence et l'intégrité des objets Java lors de leur création.

```
public class Animal {  
    private String espece;  
    public Animal (String espece)  
    {this.espece=espece;}  
}
```

```
public class Chien extends Animal {  
    private enum Race {BOXER , CANICHE ,  
    DOGUE} ;  
    private Race race ;
```

```

public Chien() { super("canide"); }
public Chien(Race race) {
    this(); /* Que se passe-t-il sans
this() ? */
    this.race=race;
}
}

```

13 Limitations

La raison principale pour laquelle une méthode surchargée ne peut pas devenir moins accessible que la méthode d'origine est liée au principe de substitution de Liskov, fondamental en programmation orientée objet. Ce principe stipule que les objets d'une classe dérivée doivent pouvoir être utilisés comme des objets de leur classe de base sans que le comportement de base de l'application soit modifié.

Si une méthode d'origine est déclarée avec un niveau d'accès plus permissif (par exemple, `public`) et que cette méthode est surchargée dans une classe dérivée avec un niveau d'accès plus restrictif (par exemple, `private`), cela violerait ce principe. En effet, un objet de la classe dérivée ne pourrait pas remplacer sans restriction un objet de la classe de base, car il ne pourrait pas fournir toutes les méthodes accessibles au niveau de la classe de base.

Cela compromettrait la capacité à utiliser le polymorphisme de manière efficace, ce qui est une pierre angulaire de la programmation orientée objet. En respectant cette règle, on garantit la cohérence et la prévisibilité du comportement des objets à travers la hiérarchie d'héritage, permettant ainsi une conception logique robuste et maintenable.

14 Le polymorphisme

Définition 2 (Polymorphisme). *Le **polymorphisme** est un mécanisme fondamental en*

programmation orientée objet qui permet à différents objets de réagir de manière spécifique à des messages identiques.

Définition 3 (Type réel). *Le type **réel** est le type de l'objet réellement créé en mémoire.*

Définition 4 (Type déclaré). *Le type **déclaré** est celui de la référence par laquelle l'objet est manipulé dans le code.*

Le type déclaré détermine les messages disponibles pour l'objet, tandis que le type réel détermine quelle action spécifique sera exécutée lorsque le message est invoqué.

```

public class Animal {
    void crier ( )
    {System.out.println("Je crie.");}
}
public class Chien extends Animal{
    void crier ( )
    {System.out.println("J'aboie");}
}
public class Chat extends Animal{
    void crier ( )
    {System.out.println("Je miaule");}
}

Animal animaux [] = {new Animal(),
new Chien() , new Chat()};

for (Animal animal:animaux )
    animal.crier();

```

15 Le transtypage

Il est possible de manipuler des objets en considérant des types différents de ceux déclarés ou réels grâce aux conversions de type, également appelées `cast`. Voici les principaux aspects à retenir :

La conversion de type permet de forcer le compilateur à traiter un objet comme appartenant

à un type différent de son type réel ou déclaré, en utilisant la syntaxe (TypeCible) objet.

L'upcast consiste à convertir un objet vers le type de sa classe mère. Cette opération est sûre car une classe fille est toujours une instance valide de sa classe mère.

En revanche, le downcast consiste à convertir un objet vers le type d'une classe fille. Cela peut entraîner une `ClassCastException` si l'objet ne peut pas être converti en cette classe fille spécifique. Il est donc important de vérifier avec `instanceof` avant de procéder à un downcast pour éviter des exceptions indésirées.

En résumé, bien que les casts soient parfois nécessaires pour manipuler des objets de manière flexible, il est crucial d'être attentif aux conversions vers des classes filles afin de maintenir la sécurité et d'éviter les erreurs d'exécution.

```
public abstract class Animal {
    abstract void crier();
}

public class Chien extends Animal {
    void mord()
{System.out.println("grr grr");}
    void crier()
{System.out.println("ouaf ouaf");}
}

public class Chat extends Animal {
    void
    crier() {System.out.println("miaou
    miaou");}
}

Animal animaux[] = {new Chat(), new
Chien( ) , new Chat ( )} ;
((Chien)animaux[1]).mord ( ) ;
/* Erreur a l ' e x e c u t i o n mais
pas a l a c o m p i l a t i o n
( ( Chien ) animaux [ 2 ] ) . mord
( ) ; */
```

16 Les classes abstraites

La spécialisation d'une classe peut souvent nécessiter la redéfinition de méthodes existantes. Il arrive que certaines méthodes n'aient pas de sens pour une classe mère en raison de la spécificité des sous-classes. Par exemple, tous les animaux peuvent émettre un cri, mais le type de cri varie selon l'espèce spécifique comme Chien, Chat, etc.

Pour gérer cette situation de manière efficace et structurée, il est recommandé de définir la méthode cri comme abstraite dans la classe mère. Cela signifie que la classe mère déclare la méthode sans en fournir l'implémentation détaillée, laissant chaque sous-classe spécialisée (comme Chien ou Chat) fournir sa propre implémentation spécifique du cri.

Les classes abstraites sont conçues spécifiquement pour être étendues et ne peuvent pas être instanciées directement. Elles servent de modèle pour d'autres classes qui étendent leur comportement et permettent de structurer le code de manière à favoriser le polymorphisme et la réutilisation du code par héritage. Ainsi, en Java, déclarer une classe comme abstraite garantit que toutes les sous-classes spécialisées implémenteront les méthodes nécessaires selon leur propre logique spécifique.

17 Les classes abstraites

En Java, lorsque vous avez des méthodes dans une classe qui sont déclarées mais n'ont pas d'implémentation concrète, vous devez les marquer avec le mot-clé `abstract`. Cela indique que la méthode est abstraite et n'a pas de corps défini dans la classe où elle est déclarée.

Les classes abstraites elles-mêmes sont des classes spéciales conçues pour servir de modèle ou de structure de base à d'autres classes. Si une classe contient au moins une méthode abstraite, elle doit être déclarée comme abs-

traite en utilisant le mot-clé `abstract`. Cela signifie que la classe n'est pas complète et ne peut pas être instanciée directement. Au lieu de cela, elle doit être étendue par d'autres classes qui fourniront des implémentations concrètes pour toutes les méthodes abstraites de la classe parente.

En résumé, en Java, les méthodes abstraites sont utilisées pour définir une interface commune que toutes les sous-classes doivent implémenter, tandis que les classes abstraites elles-mêmes fournissent cette interface et peuvent contenir à la fois des méthodes abstraites et des méthodes concrètes avec une implémentation.

```
public abstract class Animal {
    public abstract void crier();

    public static void
    faireCrier(Animal[] animaux) {
        for(Animal a:animaux)
            a.crier();
    }
}

public abstract class Bovin extends
    Animal {
    public abstract void ruminer() ;
}

public class Vache extends Bovin {
    public void crier()
    {System.out.println("Je meugle.");}
    public void ruminer()
    {System.out.println("Je rumine.");}
}

Animal.faireCrier(new Vache[]{new
    Vache()});
```

18 Les interfaces

Lorsqu'on définit une classe et qu'on utilise l'héritage, on peut définir à la fois l'état interne

des instances (attributs) et leur comportement (méthodes), souvent en les spécialisant pour des utilisations spécifiques. Cependant, cette approche impose une relation de type « est un », ce qui signifie qu'une classe fille est une spécialisation de sa classe mère.

Parfois, il est nécessaire d'imposer à des objets, qui sont des instances de classes sans relation d'héritage directe, de se conformer à un ensemble commun de comportements. Pour résoudre ce besoin, Java, tout comme UML, propose la notion d'interface. Une interface en Java définit un contrat que les classes peuvent choisir de suivre (ou « implémenter »). Elle spécifie les méthodes que les classes doivent implémenter, mais ne fournit pas d'implémentation concrète de ces méthodes. Cela permet à des classes différentes, sans lien d'héritage direct, de partager un comportement commun sans partager d'implémentation de classe.

19 Les interfaces

Définition 5. En Java, une **interface** définit un contrat sous la forme d'un ensemble de signatures de méthodes. Les classes peuvent choisir de suivre ce contrat en implémentant l'interface correspondante.

Une interface en Java est déclarée à l'aide du mot-clé `interface`. Elle ressemble à une classe qui ne contiendrait que des méthodes abstraites et publiques, c'est-à-dire des méthodes déclarées mais non implémentées. Les interfaces servent principalement à définir un contrat que les classes peuvent choisir de suivre en implémentant toutes les méthodes spécifiées.

Les interfaces en Java peuvent étendre d'autres interfaces, ce qui permet de définir une hiérarchie d'interfaces. Cette relation d'héritage entre interfaces peut être multiple, c'est-à-dire

qu'une interface peut étendre plusieurs autres interfaces, lui permettant d'hériter de leurs méthodes abstraites et constantes. Cela permet de créer des ensembles de comportements réutilisables et facilite la mise en œuvre de polymorphisme au sein du code Java.

```
public interface TrucEmpruntable {
    public void emprunter ( ) ;
    public void rendre ( ) ;
}

public class Voiture implements
TrucEmpruntable{
    boolean disponible = true ;
    public void emprunter ( )
{disponible = false ;}
    public void rendre ( ) {disponible
= true ;}
}

public class Stylo implements
TrucEmpruntable {
    boolean emprunte = false ;
    public void emprunter ( ) {emprunte
= true ;}
    public void rendre ( ) {emprunte
= false ;}
}

TrucEmpruntable trucs[] = {new
Stylo(), new Voiture()};
for(TrucEmpruntable
trucEmpruntable:trucs)
trucEmpruntable.emprunter();
```