POO – l'héritage

Table des matières

L'héritage

Introduction :

Une classe peut être étendue en une sous-classe. Cette sous-classe pourra être utilisée partout où la classe de départ pouvait l'être.

Pour définir une sous-classe on utilise la clause `extends`:

```
public class CompteAVue extends Compte {

      ...
```

Une telle sous-classe hérite de tout ce qui se trouve dans la classe d'origine appelée super-classe, classe de base, classe parent ou classe mère. La sous-classe est également appelée classe dérivée, classe enfant, classe fille.

L'héritage signifie deux choses :

- La première et la plus importante est que la sous-classe hérite du contrat de la classe parent. On veillera donc, en implémentant cette sous-classe, à respecter ce contrat.
- La deuxième est l'héritage de l'implémentation. La classe enfant hérite de toutes les méthodes et attributs accessibles et est en fait constitué de tous les attributs de la classe de base. Ceux qui sont privé (`private`) ou package () ne sont pas accessibles mais existent néanmoins. Les constructeurs ne sont jamais hérités.

En Java, toute classe hérite de la classe `Object`.

protected :

Un dernier mode d'accès est introduit. Si un attribut ou une méthode de la classe de base est déclaré `protected`, il est accessible dans les autres classes du même package ainsi que dans toutes les sous-classes, même si elles ne sont pas situées dans le package de la classe parent.

Constructeurs :

Rappelons que les constructeurs d'une classe peuvent s'appeler explicitement les uns les autres via l'instruction `this(...)` qui doit être placée en première ligne de code.

Quand un objet est construit, ses attributs sont d'abord initialisés aux valeurs par défaut, puis aux valeurs indiquées dans leur initialiseur, puis enfin aux valeurs indiquée dans le constructeur.

Une sous-classe d'une classe de base peut appeler explicitement un constructeur de la classe parent par l'instruction :

```
      super(...);
```

Cette instruction doit être la première du constructeur. C'est le constructeur de la classe parent qui est seul à même d'initialiser correctement les attributs de cette classe de base.

Si on n'appelle pas `super(...)` directement, on peut le faire indirectement en appelant un autre constructeur de la classe enfant via `this(...)` et laisser à ce constructeur le travail d'appeler `super(...)` (ou `this(...)` et ainsi de suite, jusqu'à appeler `super(...)`).

Si en bout de chaîne d'appel à `this(...)`, il n'y a pas d'appel à `super(...)`, Java effectuera d'office un appel à `super()` sans paramètres. Si la classe parent n'a pas de constructeur sans paramètres, il y aura erreur de compilation.

Ordre des initialisations :

L'ordre des initialisations est le suivant :

1. invoquer le constructeur de la superclasse pour construire l'objet en tant qu'instance de la classe parent.
2. initialiser les attributs propres à la sous-classe à l'aide des initialiseurs.
3. exécuter le reste du corps du constructeur de la sous-classe

Exemple :

```
class Parent {
	protected int a = 1;
	protected int b;
	public Parent () {
		b = a;
	}
	public String toString() {
		return a + " " + b;
	}
}

class Enfant extends Parent {
	protected int c = 3;

	public Enfant() {
		b += c;
	}

	public String toString() {
		return super.toString() + "  " + c;
	}
}
```

A l'appel de `new Enfant()`, on les étapes suivantes :

Etape	action	a	b	c
1	valeurs par défaut	0	0	0
2	appel de `Enfant()`	0	0	0
3	appel de `Parent()` = `super()`	0	0	0
4	appel du constructeur d'`Object`	0	0	0
5	initialiseur de `Parent`	1	0	0
6	constructeur de `Parent`	1	1	0
7	initialiseur `d'Enfant`	1	1	3
8	constructeur `d'Enfant`	1	4	3

Le principe de substitution :

Un objet de la classe enfant peut être utilisé partout où on utilise un objet de la classe parent.

Déclaration et définition :

En particulier lors d'une déclaration

```
Parent o = new Parent();
```

on peut remplacer l'objet de classe Parent par un objet de classe Enfant :

```
Parent o = new Enfant();
```

Nous dirons dans ce cas que Parent est la classe de déclaration de l'objet et Enfant sa classe de définition:

On a alors le principe suivant :

> Sur un objet on ne peut appeler que les méthodes déclarées dans sa classe de déclaration.

Redéfinition (Overriding) et principe de polymorphisme :

Une méthode héritée peut être redéfinie dans la classe enfant. Une méthode redéfinie aura exactement la même signature que la méthode héritée ainsi que le même type pour la valeur de retour. Ne

confondez pas redéfinition (overriding) et surcharge (overloading). Une méthode est surchargée si on définit une autre méthode de même nom mais qui n'a pas la même signature (pas la même liste de paramètres).

Si une méthode est redéfinie, elle aura un comportement différent dans la classe enfant que dans la classe parent. Si en application du principe de substitution, on remplace dans du code, un objet de la classe parent par un objet de la classe enfant, ce sera la méthode de la classe enfant qui sera appelée. Ainsi si j'ai le code :

```
Parent p = new Parent();
System.out.println(p);
```

j'aurai en sortie :

```
1 1
```

Si à présent je remplace l'appel au constructeur de `Parent` par un appel au constructeur d'`Enfant` :

```
Parent p = new Enfant();
System.out.println(p);
```

j'aurai en sortie :

```
1 4 3
```

En effet, dans la classe `Enfant`, la méthode `toString()` a été redéfinie.

Ceci nous conduit au principe de polymorphisme :

Un objet de la classe enfant peut être utilisé partout où on utilise un objet de la classe parent mais le comportement sera peut-être différent.

Cela nous amène à préciser notre principe :

> Sur un objet on ne peut appeler que les méthodes déclarées dans sa classe de déclaration mais les méthodes qui seront effectivement appelées seront celles de sa classe de définition.

Méthodes et classes `final` :

Nous avons vu qu'un attribut pouvait être déclaré `final`. Cela rend l'attribut immuable au sein de l'objet. En réalité, le mécanisme `final` est très général : une variable locale ou un paramètre peut aussi être déclaré `final`, avec la même signification d'immuabilité. Remarquons toutefois que si on déclare `final` un attribut, une variable ou un paramètre référençant un objet, la référence est non modifiable mais pas le contenu de l'objet!

Une méthode peut également être déclarée `final`. Cela signifie que cette méthode ne peut pas être redéfinie dans une sous-classe.

Enfin une classe peut être déclarée `final`. Il est alors impossible de définir des sous-classes de cette classe. C'est souvent une restriction très forte. Il est parfois préférable de déclarer toutes les méthodes de la classe, `final`, afin qu'on ne puisse les modifier tout en permettant d'ajouter du comportement à cette classe en définissant de nouvelles méthodes dans une sous-classe.

Les classes abstraites :

Il arrive parfois qu'une superclasse ne désire pas préciser l'implémentation de l'une ou l'autre méthode: elle peut le faire en marquant la méthode `abstract` : une telle méthode est appelée méthode abstraite et sera déclarée comme dans l'exemple ci-après :

```
public abstract void méthode(String argument);
```

Si une méthode d'une classe est `abstract`, la classe elle-même devra être déclarée `abstract`. Une classe peut toutefois être déclarée `abstract` sans qu'aucune méthode ne le soit. Une classe ainsi déclarée sera dite classe abstraite.

Quand une classe est abstraite, il est impossible de définir des objets de cette classe. Autrement dit, si on a

```
public abstract class Abstraite {
      ...
```

on peut déclarer

```
Abstraite objet;
```

mais il est interdit de faire

```
objet = new Abstraite(); // INTERDIT
```

Une classe abstraite sera donc nécessairement sous classée :

```
public class Concrete extends Abstraite {
      ...
```

et on pourra définir

```
Abstraite objet = new Concrete();
```

Une classe abstraite peut néanmoins avoir des constructeurs. Ceux-ci servent uniquement à être appelés dans les sous-classes :

```
public Concrete(String nom) {
      super(nom);
      ...
```

Pour qu'une sous-classe d'une classe abstraite puisse ne pas être, elle aussi, déclarée abstraite, il faut qu'elle implémente toutes les méthodes abstraites de sa classe parent.

Un exemple :

```
public abstract class Compte {
    private final String titulaire;
    private final String numéroDeCompte;
    private double solde = 0;

    public Compte( String titulaire, String numéroDeCompte ) {
        this.titulaire = titulaire;
        this.numéroDeCompte = numéroDeCompte;
    }
    public Compte( String titulaire, String numéroDeCompte,
                                                double montantInitial ) {
        this(titulaire, numéroDeCompte);
        this.solde = montantInitial;
    }

    private void transaction( double montant ) {
        this.solde += montant;
    }
    public final boolean  depot( double montant ) {
        if ( montant > 0 ) {
            transaction(montant);
            return true;
        }
        return false;
    }
    public final boolean retrait( double montant) {
        if ( montant > 0)
            if (getSolde() - montant >= getSeuil()){
                transaction(-montant);
                return true;
            }
        return false;
    }
    public boolean virement(double montant, Compte autre) {
        return false;
    }
```

```
    public String getTitulaire()[1] {
        return this.titulaire;
    }
    public String getNuméroDeCompte()[2] {
        return this.numéroDeCompte;
    }
    public double getSolde() {
        return this.solde;
    }
    public double getSeuil() {
        return 0;
    }
    public abstract String getType();
    public boolean setSeuil(double seuil) {
        return false;
    }
}

public class CompteAVue extends Compte {
    public static final String VUE = " à vue";
    private double seuil = 0;

    public CompteAVue( String titulaire, String numéroDeCompte ){
        super(titulaire, numéroDeCompte);
    }
    public CompteAVue( String titulaire, String numéroDeCompte,
                                                double montantInitial ){
        super(titulaire, numéroDeCompte, montantInitial);
    }
    public CompteAVue( String titulaire, String numéroDeCompte,
                                double montantInitial, double seuil ){
        super(titulaire, numéroDeCompte, montantInitial);
        this.seuil = seuil;
    }

    public boolean virement( double montant, Compte c ) {
        if (retrait(montant))
            return c.depot(montant);
        return false;
    }

    public String getType() {
```

[1] Rappelons que la référence vers l'attribut `titulaire` est non modifiable car il est déclaré `final`. Par contre, le contenu de l'objet renvoyé par `getTitulaire()` pourrait, lui, être modifié. Ce n'est cependant pas le cas ici car String est une classe immuable.

[2] Même remarque que pour la méthode `getTitulaire()`.

```
        return VUE;
    }
    public double getSeuil(){
        return this.seuil;
    }
    public boolean setSeuil( double seuil ){
        this.seuil = seuil;
        return true;
    }
}

public class CompteDEpargne extends Compte {
    public static final String EPARGNE = " d'épargne";

    public CompteDEpargne( String titulaire, String numéroDeCompte ){
        super(titulaire, numéroDeCompte);
    }
    public CompteDEpargne( String titulaire, String numéroDeCompte,
                                                    double montantInitial ){
        super(titulaire, numéroDeCompte, montantInitial);
    }

    public String getType() {
        return EPARGNE;
    }
}
```

Redéfinition et modificateurs :

Si une méthode héritée est redéfinie dans une classe enfant quels modificateurs sont permis pour cette méthode? Par exemple si une méthode est `protected`, doit-elle rester `protected` dans la classe enfant?

Le tableau suivant donne un début de réponse à cette question : Il indique les modificateurs permis.

méthode originale (parent)	**méthode redéfinie (enfant)**	**commentaire**
(package)	(package)	si la classe enfant est dans le même package
(package)	`protected`	si la classe enfant est dans le même package
(package)	`public`	si la classe enfant est dans le même package
`protected`	`protected`	

protected	public	
public	public	
abstract	abstract	la sous-classe est aussi abstraite
abstract		
abstract	final	
	abstract	une méthode redéfinie peut devenir `abstract` même si elle ne l'était pas dans la classe parent. La sous-classe devient abstraite
final	INTERDIT	une méthode `final` ne peut pas être redéfinie
	final	une méthode qui ne l'était pas peut devenir `final` dans une sous-classe

Héritage et static :

Une méthode ou un attribut `static` n'est jamais hérité. Une telle méthode ne peut donc pas être redéfinie. Définir une méthode de même signature masque la méthode de la classe parent. Une telle méthode ou un tel attribut `static` reste cependant accessible puisqu'il est appelé en le faisant précéder du nom de la classe qui l'a définit. De même une méthode `static` ne peut pas être abstraite.

Masquage et accès aux membres masqués via `super` :

On ne peut redéfinir que les méthodes dont on hérite, c'est à dire les méthodes qui sont accessibles dans la sous-classe (normalement `public` et `protected`, également package, si la sous-classe est dans le même package) Si on définit une méthode ayant la même signature qu'une méthode non accessible ou si on définit un attribut portant le même nom qu'un attribut de la classe de base, on parle de masquage. Un attribut ou une méthode masqué peut être accédé (s'il est accessible) via `super`.

Il faut toutefois remarquer que contrairement aux méthodes, pour accéder à un attribut, c'est toujours la classe de déclaration qui est utilisée :

> Sur un objet on ne peut accéder qu'aux attributs déclarés dans sa classe de déclaration et les attributs effectivement accédés seront ceux de la classe de déclaration.

Si une méthode non accessible (donc non héritée) est masquée, elle n'en devient pas pour autant polymorphique. Si elle est appelée (indirectement forcément, puisque non accessible) sur un objet déclaré de classe de base, c'est la méthode de la classe de base qui est appelée.

Exemple de masquage :

Considérons les classes suivantes en notant que les deux premières sont dans le même package et la troisième pas :

```
package masque;

public class Masque {
	private int a;
	int b;
	protected int c;
	public int d;

	public Masque() {	a = 1; b = 2; c = 3; d = 4; }

	public void tout() { setA(); setB(); setC(); setD();   }

	private void setA() { a = 5; }
	void setB() { b = 6; }
	protected void setC() { c = 7; }
	public void setD() { d = 8; }

	private int getA() { return a; }
	int getB() { return b; }
	protected int getC() { return c; }
	public int getD() { return d; }

	public void affiche() {
		System.out.println(a + " " + b + " " + c + " " + d);
		System.out.println(getA() + " " + getB() + " " + getC() + "
"
                                                                  +
getD());
	}
}

package masque;

public class SousMasque extends Masque {
	private int a;
	int b;
```

```
      protected int c;
      public int d;

      public SousMasque() { a = 11; b = 12; c = 13; d = 14; }

      private void setA() { a = 15; }
      void setB() { b = 16; }
      protected void setC() { c = 17; }
      public void setD() { d = 18; }

      public int getA() { return a; }
      public int getB() { return b; }
      public int getC() { return c; }
      public int getD() { return d; }
}

package sousmasque;

import masque.Masque;

public class SousMasque extends Masque {
      private int a;
      int b;
      protected int c;
      public int d;

      public SousMasque() { a = 11; b = 12; c = 13; d = 14; }

      private void setA() { a = 15; }
      void setB() { b = 16; }
      protected void setC() { c = 17; }
      public void setD() { d = 18;     }

      public int getA() { return a; }
      public int getB() { return b; }
      public int getC() { return c; }
      public int getD() { return d; }
}
```

et exécutons le programme qui suit :

```
package testmasque;

import masque.*;
```

```
public class TestMasque {
      public static void main(String[] args) {
            Masque m = new Masque();
            Masque s = new masque.SousMasque();
            Masque ss =  new sousmasque.SousMasque();
            System.out.println("Sur Masque : ");
            m.tout();
            m.affiche();
            System.out.println();
            System.out.println("Sur SousMasque (même package): ");
            s.tout();
            s.affiche();
            System.out.println();
            System.out.println("Sur SousMasque (autre package): ");
            ss.tout();
            ss.affiche();
      }
}

On aura en sortie :
Sur Masque :
5 6 7 8
5 6 7 8

Sur SousMasque (même package):
5 2 3 4
5 16 17 18

Sur SousMasque (autre package):
5 6 3 4
5 6 17 18
```

Les limitations de `protected` :

Si une méthode ou un attribut est déclaré `protected` dans une classe `A`, il est accessible dans la classe `A` (bien sûr) et dans ses sous-classes. En réalité, si dans une sous-classe `B` de `A`, on veut accéder au membre `protected` via un objet de classe `C` (sous-classe de `A`), il faut que `C` soit `B` ou une sous-classe de `B`! Tant que dans `B` on accède directement au membre `protected`, on le fait via `this` et la règle est respectée. Maintenant, si `B` est dans le même package, il n'y a aucun problème d'accès puisque l'accès `protected` implique l'accès package. Cette restriction n'a donc lieu que si `B` n'est pas dans le même package que `A`.

Exemple :

Dans la classe `Masque` ci avant, rajoutons :

```
protected void testProtected() {
      tout();
      affiche();
}
```

Dans `masque.SousMasque`, on ajoute :

```
public void testProtected(Masque autre) {
      autre.testProtected();
}

public void testProtected(SousMasque autre) {
      autre.testProtected();
}
```

Dans `sousmasque.SousMasque`, on essaye d'ajouter :

```
public void testProtected(SousMasque autre) {
      autre.testProtected();
}

public void testProtected(Masque autre) {
      autre.testProtected();
}

public void testProtected(AutreSousMasque autre) {
      autre.testProtected();
}

public void testProtected(masque.SousMasque autre) {
      autre.testProtected();
}
```

On constate que les trois dernières méthodes ne compilent pas. Ceci est du au fait que le paramètre n'est pas de classe `sousmasque.SousMasque` ni d'une de ses sous-classes.

L'héritage multiple n'est pas permis :

En Java, on ne peut hériter que d'une seule classe. Autrement dit, le mot clé `extends` ne peut être suivi que du nom d'une seule classe.

La classe Object :

Toute classe `extends` la classe `Object`. Il ne s'agit pas d'héritage multiple car si une classe `A extends B`, alors `A` héritera `d'Object` parce que `B extends Object`. Il serait donc mieux de dire, toute classe qui n'hérite pas d'une autre classe (différente d'`Object`) hérite d'`Object`.

Donc si on écrit

```
public class C {
```

c'est équivalent à

```
public class C extends Object {
```

Mais de quoi hérite toute classe de la classe `Object`? Dans la classe `Object` on trouve les méthodes publiques suivantes :

```
public boolean equals(Object obj);
```

Par défaut ne renvoie `true` que si `obj` est `this` (i.e. `obj` et `this` référencent le même objet). Les sous-classes ont donc souvent intérêt à redéfinir cette méthode.

```
public int hashcode();
```

La valeur de hashing renvoyée par cette méthode est utilisée dans les `Hashtable`, `HashMap`, `TreeMap`. Il est requis de redéfinir `hashcode()` si on a redéfini `equals()`. En effet, deux objets déclarés égaux par `equals()` doivent avoir la même valeur de `hashcode()`.

```
public String toString();
```

Par défaut renvoie une `String` formée du nom de la classe suivi d'un `@` suivi de la valeur de `hashcode()` de l'objet.

```
protected Object clone() throws CloneNotSupportedException

public final Class getClass();

protected void finalize() throws Throwable
```

Nous aborderons les trois dernières méthodes en temps voulu.