



POO : Séance 5

Héritage (Episode 2)



1^{er} semestre – Année universitaire 2024/2025

Rappelez-vous

- L'héritage permet de dériver une nouvelle classe (classe fille) à partir d'une classe existante (classe mère).
- La classe fille hérite tous les attributs et les méthodes (publics et privés) de la classe mère.
- L'accès aux attributs et méthodes privés hérités ne se fera que via des méthodes publiques.
- La classe fille peut avoir aussi ses propres attributs et ses propres méthodes.

Peut-on modifier des attributs ou des méthodes hérités pour rendre leur utilisation plus appropriée aux objets hérités?

La classe EtreHumain

```
public class EtreHumain
{
    //Déclaration des attributs
    private String nom;
    private int age;
    private String profession;

    //Déclaration des méthodes
    public EtreHumain (String leNom, int lAge, String laProfession)
    {
        nom = leNom;
        age = lAge;
        profession = laProfession;
    }

    public void sePresenter ()
    {
        System.out.println(" Mon nom est " + nom );
        System.out.println(" \n Mon age est " + age);
        System.out.println(" \n Ma profession est " + profession);
    }
}
```

La classe Etudiant

```
public class Etudiant extends EtreHumain
{

    private String filiere;

    public Etudiant(String leNom, int lAge, String laFiliere)
    {
        super(leNom, lAge, " Etudiant ");
        filiere = laFiliere;
    }
    public String quelleFiliere()
    {
        return filiere;
    }
}
```

Utilisation

```
public class test
{
    public static void main (String [] args)
    {
        EtreHumain E1 = new EtreHumain (" Mohamed ", 37, " Medecin ");
        E1. sePresenter ();
        Etudiant E2 = new Etudiant (" Ali ", 19, " Gestion ");
        E2. sePresenter ();
        System.out.println(" \n Ma filière est " + E2.quelleFiliere() );
    }
}
```

Mon nom est Mohamed
Mon age est 37
Ma profession est Medecin

Mon nom est Ali
Mon age est 19
Ma profession est Etudiant
Ma filière est Gestion

On aurait pu redéfinir
la méthode sePresenter()
dans la classe Etudiant de
manière à ce qu'elle affiche
directement les différentes
informations relatives à un
étudiant

Redéfinition de la méthode sePrésenter()

```
public class Etudiant extends EtreHumain  
{
```

```
    private String filiere;  
    public Etudiant(String leNom, int lAge, String laFiliere)  
    {  
        super(leNom, lAge, " Etudiant ");  
        filiere = laFiliere;  
    }
```



```
public void sePresenter ()  
{System.out.println(" Mon nom est " + nom );  
System.out.println(" \n Mon age est " + age);  
System.out.println(" \n Ma profession est " + profession);  
System.out.println(" \n Ma filière est " + filiere);  
}
```

```
public void sePresenter ()  
{super.sePresenter();  
System.out.println(" \n Ma filière est " + filiere);  
}  
public String quelleFiliere()  
{  
    return filiere;  
}
```

Invocation de la méthode
héritée sePresenter() de la
classe EtreHumain

Utilisation

```
public class test  
{
```

```
public static void main (String [] args)  
{
```

```
    EtreHumain E1 = new EtreHumain (" Mohamed ", 37, " Medecin ");
```

```
    E1. sePresenter ();
```

Invocation de la méthode sePresenter() de
la classe EtreHumain

```
    Etudiant E2 = new Etudiant (" Ali ", 19, " Gestion ");
```

```
    E2. sePresenter ();
```

Invocation de la méthode héritée
sePresenter() de la classe Etudiant

```
}}
```

Mon nom est Mohamed
Mon age est 37
Ma profession est
Medecin

Mon nom est Ali
Mon age est 19
Ma profession est
Etudiant
Ma filière est Gestion

Redéfinition des méthodes

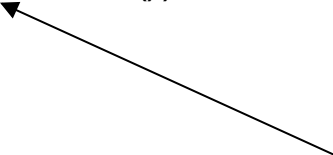
- ❖ Une sous classe peut redéfinir les méthodes qu'elle hérite et les rendre plus spécifiques à ses objets, tout en gardant le nom et les paramètres de la méthode de la super classe.
- ❖ L'utilisation d'une méthode redéfinie par un objet de la même classe permet d'utiliser cette nouvelle définition et non pas celle de la super classe.
- ❖ Le type de l'objet qui exécute la méthode détermine quelle version de la méthode est invoquée.

Règles de la redéfinition

- ❖ Les attributs et les méthodes de la classe mère sont invoqués à l'aide du mot clé super.

```
super.nom_methode()
```

```
return( 2 * super.quelMontant());
```



Invocation de la méthode
quelMontant() de la classe mère

Classe ObjetPostal

```
public class ObjetPostal
{
    //attributs
    private int poids;
    private boolean recommande;
    private float tarif;
    //méthodes
    public ObjetPostal (int lePoids, float leTarif)
    {
        poids = lePoids;
        recommande = false;
        tarif = leTarif;
    }
    public int Get_poids()
    {
        return poids;
    }
    public float queltarif()
    {
        return tarif;
    }
}
```

Classe Lettre

```
public class Lettre extends ObjetPostal  
{
```

```
    //attributs
```

```
    private boolean urgent;
```

```
    //méthodes
```

```
    public Lettre (int lePoids, float leTarif)  
    {
```

```
        super(lePoids, leTarif);  
        urgent = false;
```

Appel du constructeur
de la classe ObjetPostal

```
    }
```

```
    public void Set_Urgent()  
    {
```

```
        urgent = true;
```

```
    }
```

```
    public float quelTarif()  
    {
```

Redéfinition de la méthode quelTarif()

```
        If (urgent) return (super. quelTarif() * 2.0);
```

```
        else return (super. quelTarif() );
```

```
    }
```

Appel de la méthode quelTarif()
de la classe ObjetPostal

```
}
```

Classe Colis

```
public class Colis extends ObjetPostal  
{
```

```
    //attributs
```

```
    private int volume;
```

```
    //méthodes
```

```
    public Colis(int lePoids, float leTarif, int leVolume)
```

```
    {
```

```
        super(lePoids, leTarif);
```

```
        volume = leVolume;
```

```
    }
```

```
    public boolean grand()
```

```
    {
```

```
        if ( volume > 500 )
```

```
            return true;
```

```
        else
```

```
            return false;
```

```
    }
```

```
    public float quelTarif()
```

```
    {
```

```
        If (grand()) return (super. quelTarif() * 5.0);
```

```
        else return (super. quelTarif() * 2.0);
```

```
    }
```

```
}
```

Appel du constructeur
de la classe ObjetPostal

Redéfinition de la méthode quelTarif()

Appel de la méthode quelTarif()
de la classe ObjetPostal

Surcharge / Redéfinition

```
public class BilletAvion
{
    private double tarif;
    public double queltarif ()
    {
        ...
    }
    public double queltarif (boolean Enfant)
    {
        ...
    }
}
```

Surcharge

BilletAvion possède
deux méthodes
queltarif (surcharge)

```
public class BilletCharter extends BilletAvion
{
    public double queltarif ( )
    {
        ...
    }
}
```

Redéfinition

BilletCharter
possède une seule
méthode
queltarif()
redéfinit.
et queltarif
(boolean Enfant)
héritée.

Surcharge / Redéfinition

```
class Maman {
    // Méthode surchargée avec un paramètre int
    void afficher(int nombre) {
        System.out.println("Nombre: " + nombre);
    }

    // Méthode surchargée avec un paramètre String
    void afficher(String texte) {
        System.out.println("Texte: " + texte);
    }
}

class Fille extends Maman {
    // Redéfinition de la méthode afficher (String texte) de la classe mère
    @Override
    void afficher(String texte) {
        System.out.println("Texte (redéfini): " + texte.toUpperCase());
    }
}

public class Main {
    public static void main(String[] args) {
        Maman maman = new Maman();
        maman.afficher(10); // Appelle Maman.afficher(int)
        maman.afficher("Bonjour"); // Appelle Maman.afficher(String)

        Fille fille = new Fille();
        fille.afficher("Bonjour"); // Appelle Fille.afficher(String) - redéfini
        fille.afficher(20); // Appelle Maman.afficher(int) - surchargé
    }
}
```

Surcharge / Redéfinition

Exécution :

Nombre: 10

Texte: Bonjour

Texte (redéfini): BONJOUR

Texte: 20

Surcharge / Redéfinition



❖ La surcharge traite des méthodes portant le même nom mais ayant des paramètres différents.



Définir la même opération de façon différente pour des données différentes : Cumuler des méthodes ayant le même nom.

❖ La redéfinition traite deux méthodes, une de la classe mère et l'autre de la classe fille avec les mêmes paramètres.



Définir la même opération de façon différente pour des objets différents : Substituer une méthode par une autre.

Notions à ne pas oublier

- Héritage
- Redéfinition
- Surcharge/Redéfinition
- super

Autres notions

- Le modificateur Protected
- La classe Object
- La méthode toString()
- La méthode equals()

Le modificateur Protected

- Modificateur de visibilité qui s'applique à un membre (attribut ou méthode) d'une classe. On dit alors que le champ est protégé.
- Il permet aussi à la méthode (attribut) d'être visible dans le code des sous-classes de la classe qui la déclare, même si elles sont dans d'autres packages.
- Visibilité protected :
 - Dans la même classe : Accessible.
 - Dans les sous-classes : Accessible, même si elles sont dans des paquets différents.
 - Dans le même paquet, mais pas en sous-classe : Accessible.
 - Dans d'autres classes (pas de relation d'héritage et pas dans le même paquet) : Non accessible.

Le modificateur Protected

```
package TP ;
// Classe de base
class Animal {
    protected String nom;

    public Animal(String nom)
    {
        this.nom = nom;
    }
    protected void parler()
    {
        System.out.println(nom + " parle");
    }
}

class Chien extends Animal {
    public Chien(String nom) {
        super(nom);
    }

    public void aboyer() {
        System.out.println(nom + " aboie !");
        parler(); // Appel de la méthode protégée
    }
}
```

```
Package TP ;
// Classe principale
public class Main
{
    public static void main(String[] args)
    {
        Chien monChien = new Chien("Rex");
        monChien.aboyer();
        System.out.println(monChien.nom);
    }
}
//
```

Exécution

Rex aboie !

Rex parle

Rex

Les attributs et les méthodes 'protected' dans la classe Animal et Chien sont accessibles dans la classe Main puisque les trois classes sont dans le même package TP.

Le modificateur Protected

```
package TP ;
// Classe de base
class Animal {
    protected String nom;

    public Animal(String nom)
    {
        this.nom = nom;
    }
    protected void parler()
    {
        System.out.println(nom + " parle");
    }
}

class Chien extends Animal {
    public Chien(String nom) {
        super(nom);
    }

    public void aboyer() {
        System.out.println(nom + " aboie !");
        parler(); // Appel de la méthode protégée
    }
}
```

```
Package TP1 ;
// Classe principale
public class Main
{
    public static void main(String[] args)
    {
        Chien monChien = new Chien("Rex");
        monChien.aboyer();
        System.out.println(monChien.nom);
    }
}
//
```

Erreur de compilation. L'attribut nom est protected donc inaccessible en dehors du package TP.

La classe **Object**

- La classe **Object** est la classe mère de toutes les classes, la superclasse universelle.
 - Toutes les classes Java étendent (héritent) de la classe **Object**.
 - En fait toute classe qui n'hérite pas explicitement d'une autre classe hérite implicitement la classe **Object**.
- Donc les deux définitions de classes suivantes sont équivalentes :
 - **public class Animal { ... } — public class Animal extends Object { ... }**

La classe Object

- La classe **Object** contient les méthodes suivantes (qui seront héritées par toutes les classes en Java) :
 - **boolean equals(Object o)** : teste si deux objets sont égaux.
 - **Class getClass()** : retourne la classe à partir de laquelle l'objet a été instancié.
 - **String toString()** : retourne une chaîne de caractère pour décrire un objet.
 - Etc.

La méthode toString()

- Cette méthode est utilisée pour représenter un objet sous forme d'une chaîne de caractères.
- Appel de la méthode toString() :

Exemple :

```
CompteBancaire CPT = new CompteBancaire (10, "Ahmed", 25000) ;  
//Appel de la méthode toString  
System.out.println("Compte:  " + CPT.toString()) ;  
//ou  
System.out.println("Compte : " + CPT) ;
```

Exécution :

```
Compte:  CompteBancaire@b82e3f203  
Nom de la classe suivi par l'adresse en hexadécimal  
de l'objet CPT
```


La méthode toString()

- Lorsqu'on définit une classe, il peut être très utile de redéfinir la méthode **toString** afin de donner une description satisfaisante des objets de cette classe.

```
public String toString()
{
    return "Compte numéro " + this.numCompte + " : proprietaire " +
        this.titulaire + ", montant " + this.montant;
}
```

```
System.out.println (CPT.toString())
Compte numéro 10   : proprietaire    Ahmed, montant  25000
```

La méthode Equals()

- La méthode equals en Java est une méthode définie dans la classe Object.
- La méthode permet de comparer deux objets pour vérifier s'ils sont égaux selon des critères spécifiques.
- Par défaut, la méthode equals vérifie si les deux objets pointent vers la même référence mémoire mais elle peut être redéfinie pour des comparaisons plus spécifiques.

Exemple1:

```
public class ExempleEquals
{
    public static void main(String[] args)
    {
        // Création de deux objets de type CompteBancaire avec des valeurs
        // identiques mais références distinctes
        CompteBancaire cpt1 = new CompteBancaire (10,"Salma",10000);
        CompteBancaire cpt2 = new CompteBancaire (10,"Salma",10000);

        // Comparaison des deux objets avec la méthode equals (par défaut)
        System.out.println(cpt1.equals(cpt2)); // Affiche false
    }
}
```

La méthode Equals()

Exemple2:


```
public class ExempleEquals
{
    public static void main(String[] args)
    {
        // Création de deux objets de type CompteBancaire avec la même
        // référence
        CompteBancaire cpt1 = new CompteBancaire (10,"Salma",10000);
        CompteBancaire cpt2 = cpt1;

        // Comparaison des deux objets avec la méthode equals (par défaut)
        System.out.println(cpt1.equals(cpt2)); // Affiche true puisque cpt2
        // pointe sur cpt1
    }
}
```

La méthode Equals()

- Comment utiliser la méthode equals() pour comparer les deux comptes CPT1 et CPT2 sachant que deux comptes sont considérés comme étant identiques s'ils ont le même solde ?

```
CompteBancaire CPT1 = new CompteBancaire (10, "Ahmed", 25000) ;  
CompteBancaire CPT2 = new CompteBancaire (12, "Sami", 25000) ;
```

CPT1.equals(CPT2);  False

- Les deux comptes sont différents selon la méthode Equals de la classe Object.
- On doit donc redéfinir la méthode Equals afin de comparer deux Comptes selon l'attribut SoldeCompte.

La méthode Equals()

```
Public class CompteBancaire
```

```
{
```

```
...
```

```
...
```

```
...
```

```
Public boolean Equals(CompteBancaire C)
```

```
{
```

```
    if(this.SoldeCompte==C. SoldeCompte)  
        return true;
```

```
    Else
```

```
        return false;
```

```
}
```

```
}
```

La méthode Equals()

Public class test

{

Public static void main (String [] args)

{

CompteBancaire CPT1 = new CompteBancaire (10, "Ahmed", 25000) ;

CompteBancaire CPT2 = new CompteBancaire (12, "Sami", 25000) ;

CPT1.equals(CPT2));



True

}

}

API en JAVA

- *API : Application Programming Interface*. C'est la librairie de Java, organisée en paquetages :
 - `java.awt` : Bibliothèque pour le fenêtrage. (Graphisme.)
 - `java.io` : Gestion des entrées et sorties. (Streams, Files, etc.)
 - `java.lang` : Les classes centrales du langage. `String`, `Throwable`, etc.
 - `java.net` : Programmation réseau (URL, Datagram), etc.
 - `java.util` : Classes utiles (`Vector`, `Date`).