CHAPITRE 4 :

L'HERITAGE

Objectifs spécifiques

1. Introduire la technique d'héritage : intérêt et notation
2. Introduire les droits d'accès d'une classe dérivée aux membres de la classe de base
3. Comprendre la construction d'un objet dérivé
4. Maîtriser la notion de redéfinition
5. Découvrir le concept de dérivations multiples

Eléments de contenu

I. L'encapsulation

II. Modificateurs de visibilité et accès

III. Surcharge des méthodes

Volume Horaire :

Cours : 4 heures 30

Travaux Pratiques : 6 heures

4.1 Le principe de l'héritage

L'héritage est un mécanisme qui facilite la réutilisation du code et la gestion de son évolution. Grâce à l'héritage, les objets d'une classe ont accès aux données et aux méthodes de la classe parent et peuvent les étendre. Les sous classes peuvent redéfinir les variables et les méthodes héritées. Pour les variables, il suffit de les redéclarer sous le même nom avec un type différent. Les méthodes sont redéfinies avec le même nom, les mêmes types et le même nombre d'arguments, sinon il s'agit d'une surcharge. L'héritage successif de classes permet de définir une hiérarchie de classe qui ce compose de super classes et de sous classes. Une classe qui hérite d'une autre est une sous classe et celle dont elle hérite est une super_classe. Une classe peut avoir plusieurs sous classes. Une classe ne peut avoir qu'une seule classe mère : il n'y a pas d'héritage multiple en java.

***Object* est la classe parente de toutes les classes en java**. Toutes les variables et méthodes contenues dans Object sont accessibles à partir de n'importe quelle classe car par héritage successif toutes les classes héritent d'Object.

4.2 Syntaxe

```
class Fille extends Mere { ... }
```

On utilise le mot clé ***extends*** pour indiquer qu'une classe hérite d'une autre. En l'absence de ce mot réservé associé à une classe, le compilateur considère la classe Object comme classe parent.

Exemple

```
//Classe de base
class graphique {
private int x, y;
graphique (int x, int y) {this.x = x ; this.y = y ;}
void affiche () {
System.out.println (`` Le centre de l'objet se trouve dans :  » + x + «  et
» + y) ;}
double surface () {return (0) ;}
} // fin de la classe graphique
//Classe dérivée1
class Cercle extends graphique {
private double rayon =1 ;
void affiche () {
Sustem.out.println (« C'est un cercle  de rayon » + rayon) ;
Super.affiche () ;}
double surface ()
{return  (rayon * 2* 3.14) ;} }
//Classe dérivée2
class Rectangle extends graphique {
private int larg, longueur ;
Rectangle ( int x, int y, int l1, int l2)
{super (x,y) ;
longueur = l1 ; larg = l2 ;}
double surface () {return (longueur*largeur) ;}
}// fin de la classe Rectangle
//Classe dérivée3
class carré extends graphique {
private double côté ;
Carré (double c, int x, int y) {
Super ( x,y) ;
Côté = c ;}
Double surface () {return (côté * côté) ;}
}// fin de la classe carré
```

Interprétation

1. Dans cet exemple, la classe Cercle hérite de la classe graphique les attributs et les méthodes, redéfinit les deux méthodes *surface* et *affiche* et ajoute l'attribut *rayon*.

2. La classe Rectangle redéfinit la méthode *surface* et ajoute les attributs *longueur* et *larg*
3. La classe carré ajoute l'attribut *côté* et redéfinit la méthode *surface*.

Remarques

Le concept d'héritage permet à la classe dérivée de :

1. Hériter les attributs et les méthodes de la classe de base.
2. Ajouter ses propres définitions des attributs et des méthodes.
3. Redéfinir et surcharger une ou plusieurs méthodes héritées.
4. Tout objet peut être vu comme instance de sa classe et de toutes les classes dérivées.
5. Toute classe en Java dérive de la classe Object.
6. Toute classe en Java peut hériter directement d'une seule classe de base. Il n'y a pas la notion d'héritage multiple en Java, mais il peut être implémenté via d'autres moyens tels que les interfaces (Chapitre suivant).

Pour invoquer une méthode d'une classe parent, il suffit d'indiquer la méthode préfixée par super. Pour appeler le constructeur de la classe parent il suffit d'écrire super(paramètres) avec les paramètres adéquats.

Le lien entre une classe fille et une classe parent est géré par le langage : une évolution des règles de gestion de la classe parent conduit à modifier automatiquement la classe fille dès que cette dernière est recompilée.

En java, il est obligatoire dans un constructeur d'une classe fille de faire appel explicitement ou implicitement au constructeur de la classe mère.

4.3 L'accès aux propriétés héritées

Les variables et méthodes définies avec le modificateur d'accès public restent publiques à travers l'héritage et toutes les autres classes.

Une variable d'instance définie avec le modificateur private est bien héritée mais elle n'est pas accessible directement, elle peut l'être via les méthodes héritées.

Si l'on veut conserver pour une variable d'instance une protection semblable à celle assurée par le modificateur private, il faut utiliser le modificateur protected. La variable ainsi définie sera hérité dans toutes les classes descendantes qui pourront y accéder librement mais ne sera pas accessible hors de ces classes directement.

Exemple1 : La classe dérivée et la classe de base sont dans le même package

```
package Formes_geo ;
        class graphiques {
        private  x, y ;
        public String couleur ;
        void affiche () {
        System.out.println (« Le centre de l'objet = ( » + x + « , » + y + « ) » ) ; }
        double surface ( ) {return (0) ;} } //fin de la classe graphiques
```

```
package Formes_geo ;
        class cercle  extends graphiques{
        double rayon ;
         void changer_centre ( int x1, int y1, double r)
        {x = x1 ; y = y1 ;
        // faux car x et y sont déclarés private dans la classe de base
        rayon =r ;
        public double surface ( ) {return (rayon * 2* 3.14) ;}
        public void affichec ()
        {affiche () ;
         // juste car le modificateur de visibilité de ce membre est freindly
        System.out.println (« Le rayon =  » + rayon  + « La couleur = «  + couleur ) ;}
        }
```

Interprétation :

Dans le premier cas, la classe dérivée est la classe de base sont dans le même package, donc la classe cercle a pu accéder aux membres (attributs et méthodes) publiques (l'attribut couleur), de même elle a pu accéder aux membres « freindly » c'est-à-dire qui n'ont pas un modificateur de visibilité (la méthode affiche), mais elle n'a pas pu accéder aux membres privés (x et y).

Exemple 2 : La classe dérivée et la classe de base ne sont pas dans le même package

```
package Formes_geo ;
        class graphiques {
        private  x, y ;
        public String couleur ;
        void affiche () {
        System.out.println (« Le centre de l'objet = ( » + x + « , » + y + « ) » ) ; }
        double surface ( ) {return (0) ;} } //fin de la classe graphiques
package FormesCirc ;
        class cercle  extends graphiques{
        double rayon ;
         void changer_centre ( int x1, int y1, double r)
        {x = x1 ; y = y1 ;  // faux car x et y sont déclarés private dans la classe de base
        rayon =r ;
        public double surface ( ) {return (rayon * 2* 3.14) ;}
        public void affichec ()
        {affiche () ; // FAUX car le modificateur de visibilité de ce membre est freindly
```

```
        System.out.println (« Le rayon =  » + rayon  + « La couleur = «  + couleur ) ;}
        }
```

Interprétation :

7. Dans le deuxième cas, la classe dérivée est la classe de base ne sont pas dans le même package, donc la classe cercle a pu accéder aux membres (attributs et méthodes) publiques (l'attribut couleur), mais, elle n'a pas pu accéder aux membres « freindly » c'est-à-dire qui n'ont pas un modificateur de visibilité (la méthode affiche) et aux membres privés (x et y).
    - ⇨ Nous avons déjà vu qu'il existe 3 types de modificateurs de visibilité publiques (« public »), privés (private) et de paquetage ou freindly (sans mention).
    - ⇨ Il existe encore un quatrième droit d'accès dit protégé (mot clé « protected »).

4.4 Construction et initialisation des objets dérivés

En java, le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet.

Si un constructeur d'une classe dérivée appelle un constructeur d'une classe de base, il doit obligatoirement s'agir de la première instruction du constructeur. Le constructeur de la classe de base est désigné par le mot clé « super ».

Exemple

```
class graphiques {
        private  x, y ;
        public graphiques (int x1, int y1) {x = x1 ; y=y1 ;}
        //autres méthodes
}
class cercle extends graphiques{
          private double rayon ;
          cercle ( int a, int b, double r) {
                    super (a, b) ;
                  //appelle le constructeur de la classe de base
                   rayon = r ; }
                  //autres méthodes
}
```

Remarque

Dans le cas général, une classe de base et une classe dérivée possèdent chacune au moins un constructeur. Le constructeur de la classe dérivée complète la construction de l'objet dérivé, et ce, en appelant en premier lieu le constructeur de la classe de base.

Toutefois, il existe des cas particuliers qui sont :

- **cas1 : La classe de base ne possède aucun constructeur**

Dans ce cas, si la classe dérivée veut définir son propre constructeur, elle peut optionnellement appeler dans la 1ère instruction de constructeur la clause « super () » pour désigner le constructeur par défaut de la classe de base.

- **Cas2 : La classe dérivée ne possède pas un constructeur**

Dans ce cas, le constructeur par défaut de la classe dérivée doit initialiser les attributs de la classe dérivée et appeler :

- Le constructeur par défaut de la classe de base si elle ne possède aucun constructeur.
- Le constructeur sans argument si elle possède au moins un constructeur.
- Dans le cas où il n' y a pas un constructeur sans argument et il y a d'autres constructeurs avec arguments, le compilateur génère des erreurs.

Exemple1 (cas1)

```
Class A {
// pas de constructeur
}
class B extends A {
public B(….)
{super() ; //appel de constructeur par défaut de A
……
}
B b = new B(…);  // Appel de  constructeur1 de A
```

Exemple2 (cas2)

```
Class A {
public A() {….} //constructeur1
public A ( int n) {…..} //constructeur2
}
class B extends A {
 // …..pas de constructeur
}
B b = new B();  // Appel de  constructeur1 de A
```

Exemple3 (cas 2)

```
Class A {
public A ( int n) {…..}  //constructeur1
}
class B extends A {
 // …..pas de constructeur
}
```

```
B b = new B();
/* On obtient une erreur de compilation car A ne dispose pas
 un constructeurs sans argument et possède un autre constructeur*/
```

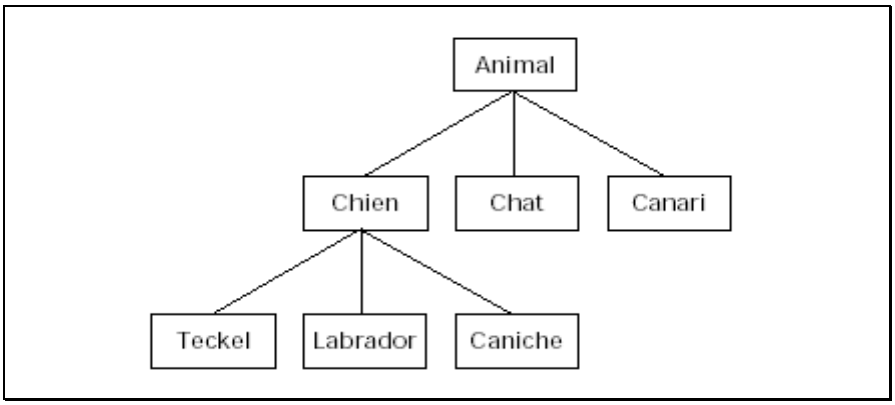
Exemple4 (cas1 et 2)

```
Class A {
//pas de constructeurs
}
class B extends A {
 //pas de constructeurs
}
B b = new B ();
// Appel de constructeur par défaut de B qui Appelle le constructeur par
//défaut de A.
```

4.5 Dérivations successives

D'une même classe peuvent être dérivées plusieurs classes différentes.

Les notions de classe de base et de classe dérivée sont relatives puisqu'une classe dérivée peut, à son tour servir de base pour une autre classe. Autrement dit, on peut rencontrer des situations telles que



✈ La classe **Caniche** est une classe descendante directement de la classe **Chien** et une classe descendante de **Animal**

✈ La notion de l'héritage multiple n'existe pas directement en Java, mais elle peut être implémentée via la notion des interfaces

4.6 Redéfinition et surcharge de membres

Nous avons déjà étudié la notion de surcharge (surdéfinition) de méthode à l'intérieur d'une même classe. Nous avons vu qu'elle correspondait à des méthodes de même nom, mais de signatures différentes. Nous montrerons ici comment cette notion se généralise dans le cadre de l'héritage : une classe dérivée pourra à son tour surcharger une méthode d'une classe ascendante. Bien entendu, la ou les nouvelles méthodes ne deviendront utilisables que par la classe dérivée ou ses descendantes, mais pas par ses ascendantes.
De même une classe dérivée peut fournir une nouvelle définition d'une méthode de même nom et aussi de même signature et de même type de retour.
=>Alors que la surcharge cumule plusieurs méthodes de même nom, la redéfinition substitue une méthode à une autre.

4.6.1 Redéfinition d'une méthode

La redéfinition d'une méthode héritée doit impérativement conserver la déclaration de la méthode parent (type et nombre de paramètres, la valeur de retour et les exceptions propagées doivent être identiques) et fournir dans une sous-classe une nouvelle implémentation de la méthode. Cette nouvelle implémentation masque alors complètement celle de la super-classe. Si la signature de la méthode change, ce n'est plus une redéfinition mais une surcharge.

Remarques

- La re-déclaration doit être **strictement identique** à celle de la super-classe : même nom, même paramètres et même type de retour.
- Grâce au mot-clé **super**, la méthode redéfinie dans la sous-classe peut réutiliser du code écrit dans la méthode de la super-classe, qui n'est plus visible autrement.

Exemple

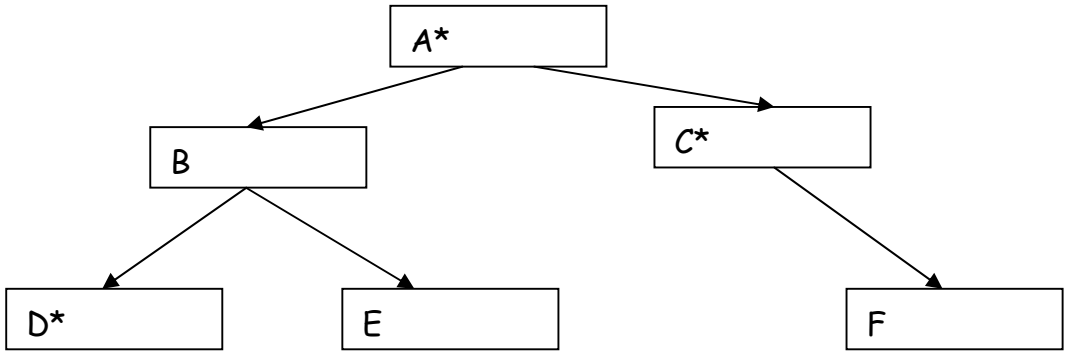
```
//Super-classe
class graphiques {
private  x, y ;
public String couleur ;
public void affiche () {
System.out.println (« Le centre de l'objet = ( » + x + « , » + y + « ) » ) ; }
double surface ( ) {return (0) ;} } //fin de la classe graphiques

//classe dérivée
class cercle  extends graphiques{
double rayon ;
 public double surface ( ) {
return (rayon * 2* 3.14) ;}   //redéfinition de  « surface »
public void affiche () //redéfinition de la méthode « affiche »
{super.affiche () ; //  appel de affiche de la super-classe
```

```
System.out.println (« Le rayon =  » + rayon  + « La couleur = «  + couleur ) ;}
}
```

4.6.2 Redéfinition de méthode et dérivations successives

Soit l'arborescence suivante :

La présence d'un astérisque (*) signale la définition ou la redéfinition d'une méthode f



L'appel de la fonction f conduira pour chaque classe à l'appel de la méthode indiquée :

Classe A : Méthode f de A

Classe B : Méthode f de A

Classe C : Méthode f de C

Classe D : Méthode f de D

Classe E : Méthode f de A

Classe F : Méthode f de C

4.6.3 Surcharge des méthodes

En Java, une méthode dérivée peut surcharger une méthode d'une classe ascendante

Exemple

```
Class A
{public void  f (int n) {…..}
…}

Class B extends A
{public void  f (float  n) {…..}
…}
A a , B b ;
int n;  float x;
a.f(n) // appelle f(int) de A
a.f(x) // Erreur,  on a pas dans A f(float)
b.f(n) // appelle f(int) de A
```

```
b.f(x) // appelle f(float) de B
```

4.6.4 Utilisation simultanée de surcharge et de redéfinition

```
Class A
{public void  f (int n) {…..}
public void  f (float n) {…..}
…}
Class B extends A
{public void  f (int  n) {…..} // redefinition  de la méthode f( int ) de A
public void  f (double  n) {…..} // Surcharge  de la méthode f de A et de  B
…}
A a , B b ;
int n , float x, double y ;
a.f(n) ;//Appel de f(int) de A
a.f(x) ;//Appel de f(float) de A
a.f(y) ; //Erreur, il n' y a pas f(double) dans A
b.f(n) ;
 //Appel de f(int) de B  car f(int) est redéfinie dans B
a.f(x) ;//Appel de f(float) de A
a.f(y) ;//Appel de f(double) de B
```

4.6.3 Règles générales de redéfinition

- Si une méthode d'une classe dérivée a la même signature q'une méthode d'une classe ascendante :
  - ✈ Les 2 méthodes doivent avoir le même type de retour.
  - ✈ Le droit d'accès de la méthode de la classe dérivée ne doit pas être moins élevé que celui de la classe ascendante.
- Une méthode statique ne peut pas être redéfinie.

4.7 Autres mots clés «final » et « super »

✈ Le mot clé « super »

- Le mot clé « super » peut être utilisé pour :

  * appeler un constructeur de la classe de base dans le constructeur de la classe dérivée. Cet appel doit être la 1ère instruction dans le constructeur de la classe dérivée.

  Syntaxe : **super (<liste des arguments>)**.

* Accéder dans une méthode d'une classe dérivée à une méthode de la super-classe (optionnel sauf dans le cas de redéfinition)

Syntaxe : **super.nom_méthode (<liste des arguments>)** ;

*Accéder dans une méthode dérivée à un attribut de la classe de base (optionnel)

Syntaxe : **super.nom_attribut**

✈ Le modificateur « final »

Cas1 : Le modificateur « final » placé devant une classe interdit sa dérivation

Exemple

```
final class A {
//Méthodes et attributs }
class B extends A   //Erreur car la classe A est déclarée finale
{//....}
```

Cas 2 : Le modificateur « final » placé devant une méthode permet d'interdire sa redéfinition

```
Class A {
final Public void f(int x) {…..}
//…Autres méthodes et attributs
}
Class B  {
//Autres méthodes et attributs
public  void f (int x) {…} // Erreur car la méthode f est déclarée finale. Pas de redéfinition
}
```

Les classes et les méthodes peuvent être déclarées finales pour des raisons de sécurité pour garantir que leur comportement ne soit modifié par une sous-classe.

4.8 Des conseils sur l'héritage

Lors de la création d'une classe « mère » il faut tenir compte des points suivants :

- la définition des accès aux variables d'instances, très souvent privées, doit être réfléchie entre protected et private ;
- pour empêcher la redéfinition d'une méthode (surcharge) il faut la déclarer avec le modificateur final. Lors de la création d'une classe fille, pour chaque méthode héritée qui n'est pas final, il faut envisager les cas suivant :
- la méthode héritée convient à la classe fille : on ne doit pas la redéfinir ;
- la méthode héritée convient mais partiellement du fait de la spécialisation apportée par la classe fille : il faut la redéfinir voir la surcharger. La plupart du temps une redéfinition commencera par appeler la méthode héritée (via super) pour garantir l'évolution du code ;
- la méthode héritée ne convient pas : il faut redéfinir ou surcharger la méthode sans appeler la méthode héritée lors de la redéfinition.