Chapitre 4 : Héritage en Java

1- Une classe B étend une classe A pour **ajouter ou modifier des services**. On peut aussi ajouter des propriétés.
2- La nouvelle classe B est appelée : classe **Dérivée**, ou classe **Fille**.
3- La classe A est appelé classe **Mère** ou classe de **Base**.
4- On dit que **B étend A ou B hérite de A**.

Contraintes :

5- Si la classe A ne dispose d'aucun constructeur (le compilateur génère un constructeur sans paramètres pour A), la classe B peut être définie sans contraintes.

Exemple :

```
class A {
   private int x, y;

   public void setX(int x) {
         this.x = x;
   }
   public int getX() {
         return x;
   }
   public int getY() {
         return y;
   }
   public void print() {
         System.out.println("(" + x + ", " + y + ")");
   }
}

Class B extends A {

}
```

➔ **Un constructeur sans paramètres est généré pour B**

6- Si A dispose d'un constructeur avec paramètres et pas de constructeur sans paramètres, alors la définition suivante est erronée :

```
Class B extends A {

}
```

7- En fait un constructeur sans paramètres est généré pour B, celui-ci appellera le constructeur sans paramètres de A qui n'existe pas. D'où l'erreur.

8- **Règle Générale** : tous les constructeurs d'une classe Fille appellent automatiquement le constructeur de la classe Mère. Cet appel est généré automatiquement par le compilateur comme première instruction de chaque constructeur.

9- Cet appel au constructeur de la classe Mère peut être réalisé explicitement à l'aide du mot clé **super** :

```
Class B extends A {
    B() {
            super() ;
            ...
    }
}
```

10- Cet appel comme première instruction signifie que si on a une classe C qui dérive de B, alors :

- Le constructeur de A est exécuté en premier lieu
- Ensuite celui de B
- Et enfin celui de C

```
Class A {
   A() {
         System.out.println("A");
   }
}
```

```
Class B extends A {
   B() {
         System.out.println("B");
   }
}
```

```
Class C extends B{
   C() {
         System.out.println("C");
   }
}
```

Une instanciation de la classe C -> l'affichage suivant

A

B

C

11- L'existence du mot clé **super** permet de dérouter l'appel à un autre constructeur avec paramètres en précisant entre parenthèses les arguments d'appel de celui-ci :

```
class A {
   private int x;
   A() {
         System.out.println("A");
   }
   A(int x) {
         This.x = x;
         System.out.println("A(x)");
   }
}
Class B extends A {
   B() {
         super(20);
         System.out.println("B");
   }
}
```

Au résultat, on aura :

A(x)

B

12- Chaque super appel la classe Mère immédiatement supérieure

```
class A {
   private int x;
   A() {
         System.out.println("A");
   }
   A(int x) {
         This.x = x;
         System.out.println("A(x)");
   }
}
```

```
Class B extends A {
   B() {
         super(20);
         System.out.println("B");
   }
}
```

```
Class C extends B {
   C() {
        super(20); // est une erreur
        System.out.println("B");
   }
}
```

*Signifie que **B** a un constructeur avec un paramètre entier, ce qui n'est pas le cas.*

13- **Remarque** : un constructeur d'une classe peut aussi appeler autre constructeur de la même classe (pour réutiliser les instructions définies dans ce dernier). On utilisera le mot clé **this**.

```
class A {
   private int x;
   A() {
        System.out.println("A");
   }
   A(int x) {
        this();
        this.x = x;
        System.out.println("A(x)");
   }
}
```

Ou encore :

```
class A {
   private int x;
   A() {
        this(20);
        System.out.println("A");
   }
   A(int x) {
        this.x = x;
        System.out.println("A(x)");
   }
}
```

14- L'instruction this() doit aussi être la première instruction du constructeur.

15- L'extension d'une classe permettra d'ajouter des services :

```
Class A {
   A() {
         System.out.println("A");
   }
   void s1() {
         ...
   }
   void s2() {
         ...
   }
}

Class B extends A {
   B() {
         System.out.println("B");
   }
   void s3() {
         ...
   }
}
```

Une instance « a » de « A » aura droit aux services : s1() et s2()

Une instance « b » de « B » aura droit aux services :

s1(), s2() (hérités) ainsi qu'au service s3().

```
A a = new A() ;
B b = new B() ;
a.s1(); a.s2(); a.s3() ;
b.s1(); b.s2(); b.s3();
```

16- L'extension d'une classe permettra de modifier des services. On parlera de la **Redéfinition**.

```
Class A {
   A() {
         System.out.println("A");
   }
   void s1() {
         System.out.println("Service s1() de A");
   }
}

Class B extends A {
   B() {
        System.out.println("B");
   }
   void s1() {
        System.out.println("Service s1() Redéfini dans B");

   }
}
```

```
A a = new A() ;
B b = new B() ;
a.s1(); ➔ "Service s1() de A";
b.s1(); ➔ "Service s1() Redéfini dans B";
```

17- Si on veut réutiliser le service s1() de la classe Mère dans la classe Fille, il est possible d'appeler le service s1 () de la classe Mère depuis la classe Fille à l'aide de l'instruction **super.s1()** :

```
Class B extends A {
   B() {
        System.out.println("B");
   }
   void s1() {
        System.out.println("Service s1() Redéfini dans B");
        super.s1();
   }
}
B b = new B() ;
b.s1(); ➔      "Service s1() Redéfini dans B";
               "Service s1() de A";
```

En résumé : le mot clé **super** désigne la classe mère. Il peut être utilisé dans deux situations différentes :

1. Pour appeler un constructeur de la classe mère : il est alors utilisé comme **première** instruction du **constructeur** de la classe fille

Syntaxe :

super(paramètres du constructeur de la classe mère) ;

Exemple :

Classe Mère :

```
class A {
   private String name;
   A(String S)
   {
        name = S;
   }
   String getName() {
        return name;
   }
}
```

Classe Fille :

```
class B extends A {

   B(String Nom) {
         super(Nom);
   }
   public static void main(String[] args) {
         B objet = new B("objet B");
         System.out.println(B.getName());
   }
}
```

2. Pour appeler une méthode de la classe mère, si celle-ci est redéfinie dans la classe fille.

Syntaxe :

super.nomDeLaMethode(paramètres) ;

Exemple :

```
class A {
  void p()
  {
    System.out.println("Méthode p() de la classe A");
  }
}

public class B extends A {

  void p()
  {
    System.out.println("Méthode p() de la classe B");
    super.p();
  }
  public static void main(String[] args) {
    B objet = new B();
    objet.p();
  }
}
```

Le programme affichera :

Méthode p() de la classe B

Méthode p() de la classe A

18- Remarque : la classe Object est la classe mère de toutes les classes sans mère. Ainsi la définition suivante :

```
class A {

   ...

}
```

Est équivalente à :

```
class A extends Object {
    ...
}
```

Résultats :

19- Résultat N° 1 : la classe **Object** se trouve en tête de l'arbre d'héritage de toutes les classes Java.
20- Résultat N° 2 : toute nouvelle classe va hériter toutes les méthodes public de la classe Object
21- Résultat N° 3 : comme exemple de méthodes, il y a la méthode **toString**() invoquée automatiquement par le compilateur lors d'une tentative d'affichage d'un objet :

```
System.out.println(objet)
```

⇔

```
System.out.println(objet.toString()) //générée par le compilateur
```

Mais aussi dans tous contexte nécessitant cette transformation : généralement la concaténation de chaines de caractères :

```
String s ;
s + objet + … ⇔ s + objet.toString() + …
```

22- La méthode toString() est une méthode public de la classe Object qui retourne une chaine de caractère ; l'objet est alors « imprimable » ou « affichable ».
23- La méthode toString() est définie par défaut (dans Object) pour retourner l'adresse de l'objet sous forme d'une chaine de caractères.
24- Il est donc possible de la redéfinir dans n'importe quelle classe pour choir le format adéquat d'afficher des objets de la classe.

Exemple :

```
class A {
    private int x, y;
    ...
    public String toString() {
        return "(" + x + ", " + y + ")") ;
    }
}
```

NECESSAIRE

25- La redéfinition d'une méthode ne doit pas réduire ses privilèges d'accès :

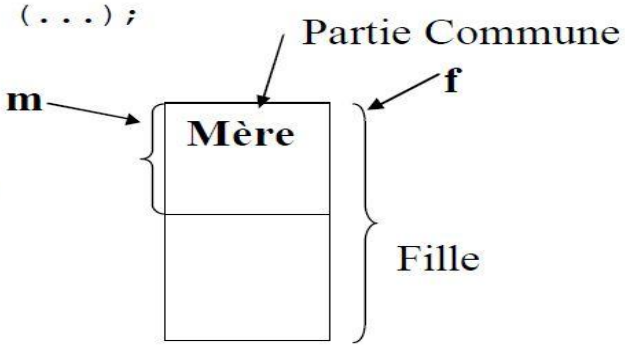
```
private void p() ; ➔(Red)➔ private, protected, "sans" , public
protected void p() ; ➔(Red)➔ protected, "sans" , public
void p() ; ➔(Red)➔ "sans" , public
public void p() ; ➔(Red)➔ public
```

26- Important : un objet de classe Fille peut être affecté à un objet de classe Mère :

L'affectation [Mère <- Fille] est possible :

```
Fille f = new Fille (...);
Mère m = f ;
```

➔ l'objet « m » aura une
Vue plus restreinte que « f »



27- Important : l'affectation [Fille <- Mère] n'est possible que :

- Si on utilise un transtypage (casting)
- Si l'objet pointe effectivement un objet fille :

Exemple :

```
Fille f1 = new Fille(…) ;
Mère m = f1 ; // l'objet mère Pointe une fille.
Fille f2 = (Fille) m ; // Possible
Mère m2 = new Mère(…) ;
Fille f3 = (Fille)m2 ;// Possible lors de la compilation (syntaxiquement).
```

Mais ERREUR lors de l'Exécution.

Remarques:

28- Le casting des objets en java permet de Retrouver l'identité réelle d'un objet
29- Pas de casting entre des objets de classes n'ayant pas de relation Mère-Fille
30- Cette relation peut ne pas être directe

Exemple :

C étend B et B étend A, on peut faire :

```
C c1 = new C(…) ;
A a = c1 ; // Mère ← Fille
C c2 = (C) a; // Fille ← Mère
```

Classe Abstraites et Interfaces

En Java, il existe 3 types d'entités qu'on peut manipuler :

1- Les classes (déjà vues)
2- Les **classes abstraites** présentées par le mot clé **abstract** :

```
abstract class NomClasse {
    ...
}
```

Dans une classe abstraite, le corps de quelques méthodes peut ne pas être défini (on déclare uniquement le prototype de la méthode). Ces méthodes sont dites des méthodes abstraites. Une méthode abstraite est aussi présentées par l'intermédiaire du mot clé abstract de la manière ci-après. C'est aux classes dérivées de redéfinir ces méthodes et de préciser leur comportement.

```
abstract class NomClasse {
    abstract type NomMéthode(parameters);
    ...
}
```

Une classe abstraite ne peut donc jamais être instanciée. Il s'agit d'une spécification devant être implémentée par l'intermédiaire d'une classe dérivée. Si cette dernière définit toutes les méthodes abstraites alors celle-ci est instanciable.

Remarque :

Une classe abstraite peut ne pas contenir des méthodes abstraites. Cependant, une classe contenant une méthode abstraite doit obligatoirement être déclarée **abstract**.

3- Les **interfaces** qui sont définies par l'intermédiaire du mot clé **interface** au lieu de **class** constituant un cas particulier des classes abstraites : d'une part, ce sont des *classes* où aucune méthodes n'est définies (uniquement le prototype de chaque méthode). D'autre part, l'extension d'une interface est appelé **implémentation** et elle est réalisé par l'intermédiaire du mot clé **implements**.

Définition d'une interface :

```
interface NomInterface {
      type1 methode1(paramètres);
      type2 methode2(paramètres);
      …
}
```

Implémentation d'une interface :

```
class NomDeClasse implements NomInterface {
 …
}
```

Remarques :

1. Une classe qui implémente une interface doit définir toutes les méthodes de l'interface
2. Une interface peut aussi contenir des attributs
3. Tous les attributs d'une interface doivent obligatoirement initialisés
4. Tous les attributs d'une interface sont **public**, **static** et **final**.