

Java : mots-clés final, static, abstract & Interfaces

1. Attributs, méthodes et classes finaux.....2
2. Attributs & méthodes statiques2
3. Méthodes & classes abstraites.....3
4. Interfaces.....5

1. Attributs, méthodes et classes finaux

Le mot-clé *final* peut s'appliquer à un attribut, à une méthode ou à une classe. Il se rajoute soit avant, soit après le modificateur private, protected ou public.

- **Un attribut final** est un attribut dont la valeur initiale ne peut pas être modifiée, autrement dit c'est une constante.
- **Une méthode finale** est une méthode qui ne peut pas être redéfinie dans une classe dérivée.
- **Une classe finale** est une classe qui ne peut pas être dérivée.

Exemple : la classe Math, dans laquelle sont définies les opérations mathématiques usuelles, est une classe finale.

```
public final class Math {
    ...
}
public class SousClasseMath extends Math {
    //IMPOSSIBLE : la classe Math ne peut pas avoir de sous-classe
}
```

Les méthodes déclarées dans une classe finale sont nécessairement finales. Dès lors, le mot-clé final n'est pas obligatoire devant les méthodes des classes finales, il est implicite.

2. Attributs & méthodes statiques

Le mot-clé *static* peut s'appliquer à un attribut ou à une méthode. Il se rajoute soit avant, soit après le modificateur (private, protected ou public).

a. Attributs statiques

Un attribut statique est un attribut qui est associé non pas à un objet mais à la classe elle-même. L'attribut statique est donc unique, quel que soit le nombre d'objets créés.

Pour l'utiliser dans une autre classe, on peut l'appliquer à la classe elle-même.

Remarquons que lorsqu'un attribut a une valeur constante et que cette valeur est la même pour toutes les instances de la classe, il doit donc être déclaré final et statique. C'est le cas, par exemple, de la constante PI de la classe Math (cf. exemple ci-dessous).

b. Méthodes statiques

Une méthode statique est une méthode qui n'est associée à aucun objet, elle définit une opération de la classe.

Pour l'utiliser dans une autre classe, on peut l'appliquer à la classe elle-même.

Exemple : dans la classe Math les attributs et les méthodes sont statiques.

```
public final class Math {
    public static final double PI = 3.14159265358979323846;
    ...
    public static double sin(double a) {...}
```

```

public static double cos(double a) {...}
public static double exp(double a) {...}
public static double sqrt(double a) {...}
...
}
  
```

Dans une classe principale, on peut utiliser les attributs et méthodes statiques de la classe Math, sans déclarer d'instance de cette classe :

```
double x = Math.sqrt(16);           //pas besoin d'instancier d'objet
```

c. Règles d'accès

- Les méthodes et les attributs statiques sont **accessibles dans toutes les méthodes** (statiques ou non) de la classe. Si leur niveau de visibilité le permet, ils sont aussi accessibles dans les méthodes des sous-classes et dans les méthodes des autres classes.
- Une méthode statique **peut accéder à des méthodes et à des attributs statiques, mais pas aux autres** méthodes et attributs de la classe (à moins d'être appliquée à un objet de la classe).

Exemple :

On a écrit une classe A :

```

public class A {
    static private int compteur = 0;           //un attribut statique est unique, quel que soit le nombre d'objets crée
    public static void modifier () {
        compteur++;                         //une méthode statique peut accéder à un attribut statique
    }
    public void afficher(){
        compteur++;                         //un attribut statique est accessible dans une méthode non statique
        System.out.println(compteur);
    }
}
  
```

puis, dans la classe principale, on a écrit les instructions suivantes :

```

A a1 = new A();
A a2 = new A();
A.modifier();          //on incrémente le compteur en appelant une méthode statique
a1.afficher();         //on incrémente et on affiche la valeur du compteur => le programme affiche 2
a2.afficher();         //on incrémente et on affiche la valeur du compteur => le programme affiche 3
  
```

3. Méthodes & classes abstraites

Le mot-clé *abstract* peut s'appliquer à **une méthode ou à une classe**. Il se rajoute soit avant, soit après le modificateur private, protected ou public.

a. Méthodes abstraites

Une méthode abstraite est une méthode sans corps (non implémentée dans la classe où elle est déclarée), que la ou les classes filles doivent implémenter.

Une méthode abstraite doit obligatoirement être déclarée protégée ou publique (sa vocation est d'être

redéfinie dans la ou les classes dérivées).

Lorsqu'une classe comporte une ou plusieurs méthodes abstraites, elle est obligatoirement abstraite.

Exemple : on a écrit une classe Appareil permettant de décrire les propriétés communes à plusieurs appareils de mesure (voltmètres, ampèremètres, oscilloscopes, etc.). Tous les appareils ont un nom et doivent pouvoir être initialisés. On a écrit une sous-classe Voltmetre décrivant des voltmètres numériques connectés à un PC. Un voltmètre permet de mesurer une tension qui, au choix, est une tension AC ou une tension DC.

```

public abstract class Appareil {           //classe abstraite
    private String nom_appareil ;
    public Appareil (String nom){
        nom_appareil=nom ;
    }
    abstract public void initialiser();    //la méthode abstraite sera implémentée dans les sous-classes
    public void afficher()
    {
        System.out.println("nom = "+nom_appareil);
    }
}

public class Voltmetre extends Appareil{
    private double tensionAC ;             //tension AC
    private double tensionDC ;             //tension DC
    private boolean choix_tensionAC ;      //choix d'une lecture AC (true) ou DC (false)
    public Voltmetre (String nom, double initAC, double initDC){
        super(nom) ;
        tensionAC=initAC ;
        tensionDC=initDC ;
        choix_tensionAC=true ;
    }
    public void initialiser(){
        //code d'initialisation du voltmètre
        //(obligatoire pour que la classe soit instanciable)
    }
    public void afficher()
    {
        //code de la méthode d'affichage
    }
    public void lire_tension()
    {
        // code de lecture d'une tension
    }
}

```

Appareil

- nom_appareil : String
- +Appareil(nom: String)
- +initialiser() : void
- +afficher() : void



Voltmetre

- tensionAC : double
- tensionDC : double
- choix_tensionAC : boolean
- +Voltmetre(nom: String, initAC: double, initDC: double)
- +initialiser() : void
- +afficher() : void
- +lire_tension() : void

Tout appareil de mesure (voltmètre, ampèremètre, etc.) doit pouvoir être initialisé, mais pas de la même façon. En déclarant une méthode initialiser abstraite dans la classe Appareil, on oblige chaque classe dérivée à implémenter la méthode.

b. Classes abstraites

Une classe abstraite est une classe dont on ne peut pas créer d'instances. Elle ne peut donc servir que de classe de base pour une dérivation.

Une classe abstraite peut bien sûr contenir des méthodes non abstraites et des attributs dont hériteront ses classes filles. Elle peut aussi contenir des méthodes abstraites.

Exemple :

Dans une classe principale, on peut déclarer une variable de type appareil, mais il sera impossible de l’instancier :

```
Appareil a ;
a = new Appareil ("Hewlett Packard");
```

En revanche, on peut instancier la classe Voltmetre qui dérive de la classe Appareil :

```
a = new Voltmetre ("Hewlett Packard",0,0); //création d'un voltmètre
```

Remarques :

- pour parler d'une méthode ou d'une classe qui n'est pas abstraite, on emploie le terme **concrète**,
- une classe peut être abstraite sans comporter de méthodes abstraites,
- lorsqu'une classe est abstraite, ses sous-classes ne pourront être concrètes que si elles implémentent toutes les méthodes abstraites de leur super-classe (sinon elles seront aussi abstraites).

4. Interfaces

Une interface est un type dans lequel on ne trouve que des méthodes abstraites et des constantes. Pour déclarer une interface, on utilise le mot-clé *interface*.

Une classe peut implémenter plusieurs interfaces, alors qu'une classe ne peut dériver que d'une seule classe abstraite ou concrète.

Pour indiquer que dans une classe on implémente les méthodes d'une interface, on utilise le mot-clé *implements*.

Exemple :

On a écrit deux interfaces et deux classes concrètes :

```
public interface Interface1{
...
}
public interface Interface2{
...
}
public class Base1{
...
}
public class Base2{
...
}
```

On peut ensuite écrire les classes suivantes :

```
public class Fille1 extends Base1{
//cette classe est une sous-classe de la classe Base1
...
}
```

```
public class Fille2 implements Interface1{
//cette classe implémente l'interface Interface1
...
}
public class Fille3 extends Base1 implements Interface1{
//cette classe est une sous-classe de la classe Base1 et elle
implémente l'interface Interface1
...
}
public class Fille4 implements Interface1, Interface2{
//cette classe implémente deux interfaces Interface1 et
Interface2
...
}
```

En revanche, on ne peut pas écrire :

```
public class Fille5 extends Base1, Base2{
//IMPOSSIBLE : une classe ne peut pas être sous-classe de
deux classes
}
```

Les méthodes d'une interface sont nécessairement publiques et abstraites. Dès lors, les mots-clés *abstract* et *public* ne sont pas obligatoires.

Les attributs d'une interface sont forcément des attributs publics statiques et finaux (des constantes). De même, les mots-clés *public*, *static* et *final* ne sont pas obligatoires.

Les méthodes d'une interface étant abstraites, les classes qui implémentent cette interface doivent obligatoirement les implémenter pour être instanciables (i.e. pour ne pas être des classes abstraites).

Exemple :

```
public interface I{  
    void f(int n);           //les mots-clés public et abstract sont implicites  
    void g();  
    int MAXI = 100;          //les mots-clés public, static et final sont implicites  
}  
public class A implements I{ //pour être instanciable, la classe A doit redéfinir f et g  
    //dans toutes les méthodes de A, on a accès au symbole MAXI :  
    // par exemple : if (i < MAXI) ....  
}
```

Dans le chapitre précédent, nous avons déjà vu plusieurs exemples d'interfaces.

L'interface *Collection* permet de décrire les méthodes communes à toutes les collections. Ses deux sous-interfaces *List* et *Set* permettent de décrire, respectivement, les méthodes communes à toutes les listes et les méthodes communes à tous les ensembles.

Nous avons vu aussi par exemple que pour définir un ordre naturel dans une classe, la classe doit implémenter l'interface *Comparable* et elle doit redéfinir la méthode abstraite *compareTo*.