Chapitre 6 : Les classes abstraites et les interfaces

Objectif général

- Comprendre le concept de classes abstraites et interfaces.

objectifs spécifiques

- Comprendre le concept de classes abstraites et leurs intérêts.
- Comprendre le concept d'interface, leurs intérêts pratiques et leurs relations avec l'héritage.

Eléments de contenu

- Les classes abstraites
- Les interfaces

I. Classes abstraites

1.1 Présentation

Une classe abstraite est une classe qui ne permet pas d'instancier des objets. Elle ne peut servir que de classe de base dans une relation d'héritage. Elle se déclare ainsi :

```
abstract class A
{................
}
```

Dans une classe abstraite on peut trouver des attributs et des méthodes comme une classe ordinaire. Mais on peut aussi trouver des méthodes dites **abstraites**, c'est-à-dire dont on ne fournit que la signature et le type de la valeur de retour et elle doit se terminer par un point virgule. Par exemple :

```
abstract class A
{
        public void f(....) // f est définie dans A
        {..................................
        }
        public abstract void g(int n); // g n'est pas définie dans A; on n'en a
```

Bien entendu, on pourra déclarer une référence de type A :

```
A a ;
```

En revanche, toute instanciation d'un objet de type A sera rejetée par le compilateur :

```
A= new A(.....) ; // erreur pas d'instanciation d'objets d'une classe abstraite
```

En revanche, si on dérive de A une classe B qui définit la méthode abstraite g :

```
class B extends A
{public void g(int n)
{ ........// ici on définit g
}}
```

On pourra alors instancier un objet de type B par new B(….) et même affecter sa référence à une référence de type A :

```
A a =new B(); // OK
```

1.2 Quelques règles

1. Dans l'en-tête d'une méthode déclarée abstraite, les noms des arguments doivent figurer (bien qu'ils ne servent à rien)

```
abstract class A
{public abstract void g(int); // erreur nom d'argument obligatoire
}
```

2. Une classe dérivée d'une classe abstraite peut ne redéfinir aucune méthode abstraite ou redéfinir quelques méthodes abstraites. Dans ce cas, elle reste simplement abstraite.

```
abstract class A
{       public abstract void f1();
        public abstract void f2(char c);
}
abstract class B extends A
{       public void f1()//définition de f1 mais pas de définition de f2
        { ………………………………
        }
}
```

1.3 Exemple

Cette technique facilite l'évolution de l'application car si une nouvelle fonctionnalité doit être disponible dans les classes dérivées, il suffit d'ajouter cette fonctionnalité dans la classe de base et de ne pas fournir son implémentation et ainsi laisser à l'utilisateur de la classe le soin de créer l'implémentation dans la classe dérivée.

Voici un exemple illustrant l'emploi d'une classe abstraite nommée affichable, dotée d'une seule méthode abstraite affiche.

Exemple

```
abstract class Affichable
{public abstract void affiche();

}

   class Entier extends Affichable {
        private int valeur;

          public Entier(int n) {
                    valeur = n;
          }

          public void affiche() {
                    System.out.println("je suis un entier de valeur " + valeur);

          }
}

    class Flottant extends Affichable {
        private float valeur;

          public Flottant(float x) {
                    valeur = x;
          }

          public void affiche() {
                    System.out.println("je suis un flottant de valeur " + valeur);

          }
}

class Test {
          public static void main(String args[]) {
                    Affichable[] tab = new Affichable[3];
                    tab[0] = new Entier(25);
                    tab[1] = new Flottant(1.25f);
                    tab[2] = new Entier(42);

                    for (int i = 0; i < 3; i++)
                              tab[i].affiche();
          }

}
```

1.4 Ce qu'il faut retenir

Ce qu'il faut retenir c'est que :

- Il n'est pas possible d'avoir un objet d'une classe abstraite ;
- Toute classe qui contient au moins une méthode abstraite est, elle aussi, abstraite ;
- Toute classe qui dérive d'une classe abstraite doit absolument définir toutes les méthodes abstraites des classes parent sinon elle sera, elle aussi, abstraite ;
- Le mot-clé abstract apparaît à la fois au début de la classe abstraite et comme préfixe de toutes les méthodes abstraites ;

II. Création d'une interface

On a vu déjà qu'on peut obliger une classe à implémenter en déclarant celle-ci avec le mot clé abstract. Si plusieurs classes doivent implémenter la même méthode, il est plus pratique d'utiliser les interfaces.

2.1 Définition d'une interface

Comme les classes, les interfaces permettent de définir un ensemble de constantes et méthodes.

Cependant les interfaces ne contiennent aucun code. On peut dire donc qu'une interface est une spécialisation complétement abstraite puisque toutes ses méthodes sont implicitement **public** et **abstract**.

Aussi toutes ses données(attributs) sont implicitement **static** et **final**.

<u>NB : le terme "interface" n'a rien à voir ici avec " les interfaces graphiques" .</u>

2.2 Déclaration d'une interface

La déclaration d'une interface est semblable à celle d'une classe et on peut éventuellement utiliser le mot clé **extends** pour introduire une relation d'héritage dans l'interface. Contrairement aux classes les interfaces autorisent l'héritage multiple.

```
 interface  NomDeLInterfaec extends AutreInterface1, AutreInterface2
{// déclaration de constantes
// déclaration des signatures de méthodes
}
```

Il est recommandé de fournir sous forme de commentaires une description du travail que devra accomplir chaque méthode et les résultats qu'elle devra fournir.

Exemple

```
// Cette interface devra être implémentée par les classes pour lesquelles un classement des instances est envisagée
interface  Classable
{// Cette méthode pourra être appelée pour comparer l'objet en cours avec l'objet reçu en paramètre
int compare(Object o) ;
//la méthode retourne
//1 si l'objet en cours est supérieur à celui reçu en paramètre
// 0 si les deux objets sont égaux
/-1 si l'objet en cours est inférieur à celui reçu en paramètre
// -99 si la comparaison est impossible

public static final int INFERIEUR=-1 ;
public static final int EGAL=0;
public static final int SUPERIEUR=1;
public static final int ERREUR=-99;
}
```

2.3 Utilisation d'une interface

Une interface permet de définir un contrat fonctionnel avec toutes les classes qui décident de l'implémenter : une classe qui implémente une interface doit impérativement concrétiser (définir) toutes les méthodes de l'interface en respectant impérativement les signatures des méthodes.

a. Syntaxe de l'utilisation d'une interface

En java, une classe ne peut pas être dérivée directement de plusieurs classes. En revanche, la notion d'interface peut être utilisée pour ce type d'héritage (héritage multiple).

Une classe annonce qu'elle implémentera d'une ou de plusieurs interfaces via le mot clé **implements**.

```
class  NomClasse  [extends  NomClasse]  [implements  NomInterface1, NomInterface2, NomInterface3,…….]
{
…………
}
```

Si une classe implémente plus d'une interface, elle doit implémenter toutes les méthodes de chacune des interfaces mentionnées dans la clause **implements** (ou alors être déclarée **abstract**).

b. Exemple d'utilisation d'une interface

Pour l'exemple de l'interface Classable c'est à l'utilisateur de cette interface de choisir les critères de comparaison. Par exemple pour le cas de la classe Personne, on peut implémenter l'interface Classable en choisissant comme critère de comparaison le nom.

```
 class Personne implements Classable
{ private String nom ;
  public String getNom(){
  return this.nom ;   }
   public int compare(Object o)
  {Personne p;
    if ( o instanceof Personne)
     { p=(Personne) o;
      }
     else
      { return Classable.ERREUR;
       }
     if (this.getNom().compareTo(p.getNom())<0)
         return Classable.INFERIEUR;
     if (this.getNom().compareTo(p.getNom())>0)
         return Classable.SUPERIEUR;
      return Classable.EGAL;
}}
```

2.4 Comparaison des interfaces avec les classes abstraites

Le corps d'une interface ressemble beaucoup à celui d'une classe abstraite avec cependant les différences suivantes :

- Une interface ne peut définir que des méthodes abstraites. Le modificateur abstract n'est pas nécessaire.
- Toutes les **méthodes** abstraites sont implicitement **publiques** même si le modificateur public est omis.
- Une interface ne peut pas définir des champs d'instance.
- Une interface peut cependant contenir des constantes déclarées **static** et **final** (tous les champs sont implicitement static et final même si les modificateurs correspondants sont omis).
- Une interface ne définit pas de **constructeurs** (on ne peut pas l'instancier et elle n'intervient pas dans le chaînage des constructeurs).

- L'avantage des interfaces est qu'il est possible d'implémenter plusieurs interfaces, or on ne peut pas hériter de plusieurs classes.