Classes abstraites

Classes abstraites

- Il peut être nécessaire au programmeur de créer une classe déclarant une méthode sans la définir (c'est-à-dire sans en donner le code). La définition du code est dans ce cas laissée aux sous-classes.

- Une telle classe est appelée **classe abstraite**.
- Elle doit être marquée avec le mot réservé abstract.
- Toutes les méthodes de cette classe qui ne sont pas définies doivent elles aussi être marquées par le mot réservé abstract.

- Une classe abstraite ne peut pas être instanciée.

Classes abstraites

- Par contre, il est possible de déclarer et d'utiliser des variables du type de la classe abstraite.

- Si une sous-classe d'une classe abstraite ne définit pas toutes les méthodes abstraites de ses superclasses, elle est abstraite elle aussi.

Classes abstraites

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

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

Dans une classe abstraite, on peut trouver classiquement des méthodes et des champs, dont héritera toute classe dérivée.

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.

Classes abstraites

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 fourni que l'en-tete
}
```

Bien entendu, on pourra déclarer une variable de type *A* :

A a ; // OK : a n'est qu'une reference sur un objet de type A ou dérivé,

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

Classes abstraites

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 fourni que l'en-tete
}
```

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 variable de type *A* :

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

Quelques règles

1) Dès qu'une classe comporte une ou plusieurs méthodes abstraites, elle est abstraite.

```
class A
{ public abstract void f() ; // OK
.....
}
```

Malgré tout, *A* est considérée comme abstraite et une expression telle que *new A(...)* sera rejetée.

2) Une méthode abstraite doit obligatoirement être déclarée *public*, ce qui est logique puisque sa vocation est d'être redéfinie dans une classe dérivée.

3) Dans l'en-tête d'une méthode déclarée abstraite, les noms d'arguments muets doivent figurer

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

Quelques règles

4) Une classe dérivée d'une classe abstraite n'est pas obligée de (re)définir toutes les méthodes abstraites de sa classe de base .
Dans ce cas, elle reste simplement abstraite (il est quant même nécessaire de mentionner *abstract* dans sa déclaration) :

```
abstract class A
{ public abstract void f1() ;
public abstract void f2 (char c) ;
.....
}
abstract class B extends A      // abstract obligatoire ici
{ public void f1 () { ..... }    // definition de f1
.....                            // pas de definition de f2
}
```

Ici, *B* définit *f1*, mais pas *f2*. La classe *B* reste abstraite,

5) Une classe dérivée d'une classe non abstraite peut être déclarée abstraite et/ou contenir des méthodes abstraites.

Intérêt des classes abstraites

Le recours aux classes abstraites facilite largement la conception orientée objet. En effet, on peut placer dans une classe abstraite toutes les fonctionnalités dont on souhaite disposer pour toutes ses descendantes :

- soit sous forme d'une implémentation complète de méthodes (non abstraites) et de champs (privés ou non) lorsqu'ils sont communs à toutes ses descendantes,

- soit sous forme d'interface de méthodes abstraites dont on est alors sûr qu'elles existeront dans toute classe dérivée instanciable.

Exemple:

programme illustrant l'emploi d'une classe abstraite nommée ***Affichable***, dotée d'une seule méthode abstraite *affiche*.

Deux **classes *Entier* et *Flottant*** dérivent de cette classe. La méthode ***main*** utilise un tableau hétérogène d'objets de type ***Affichable*** qu'elle remplit en instanciant des objets de type ***Entier* et *Flottant***.

Intérêt des classes abstraites

```
abstract class Affichable
{ abstract public void affiche() ;
}
class Entier extends Affichable
{ public Entier (int n)
{ valeur = n ;
}
public void affiche()
{ System.out.println ("Je suis un entier de
valeur " + valeur) ;}

private int valeur ;}

class Flottant extends Affichable
{ public Flottant (float x)
{ valeur = x ;
}
public void affiche()
{ System.out.println ("Je suis un flottant de
valeur " + valeur) ;
}
private float valeur ;}
```

```
public class Tabhet3
{ public static void main (String[] args)
{ Affichable [] tab ;
tab = new Affichable [3] ;
tab [0] = new Entier (25) ;
tab [1] = new Flottant (1.25f) ; ;
tab [2] = new Entier (42) ;
int i ;
for (i=0 ; i<3 ; i++)
tab[i].affiche() ;
}
}
```

```
Je suis un entier de valeur 25
Je suis un flottant de valeur 1.25
Je suis un entier de valeur 42
```

Classe d’interfaces

Les interfaces

- une classe abstraite permettait de définir dans une classe de base des fonctionnalités communes à toutes ses descendantes…

- Si l'on considère une classe abstraite n'implantant aucune méthode et aucun champ (hormis des constantes), on aboutit à la notion **d'interface.**

- Une interface définit les en-têtes d'un certain nombre de méthodes, ainsi que des constantes.

Les interfaces

- une classe pourra implémenter plusieurs interfaces (alors qu'une classe ne pouvait dériver que d'une seule classe abstraite),

- la notion d'interface va se superposer à celle de dérivation, et non s'y substituer,

- les interfaces pourront se dériver,

- on pourra utiliser des variables de type interface

A quoi ça sert ?

- Les classes '**interfaces**' servent à créer des comportements génériques:
  - si plusieurs classes doivent obéir à un comportement particulier, on créé une interface décrivant ce comportement, on est la fait implémenter par les classes qui en ont besoin.
  - Ces classes devront ainsi obéir strictement aux méthodes de l'interface (nombre, type et ordre des paramètres, type des exceptions), sans quoi la compilation ne se fera pas.

A quoi ça sert ?

- Les interfaces sont très similaires aux classes (une interface par fichier .class...), à la différence que les interfaces ne peuvent être instanciées : elles sont *implémentées* dans la classe via le mot-clé **`implements`**.

Interface

- Classe particulière qui ne peut comprendre
  - que des méthodes abstraites
  - ou des constantes
- Différences par rapport à une classe abstraite
  - **toutes** les méthodes sont abstraites
  - pas d'attribut
- Syntaxe : `[visibilité] interface nom_interface { …}`
- Une interface exprime un **comportement**

ATTENTION : ne pas confondre les *classes d'interface* avec *l'interface graphique (IHM)*, ni avec la notion *d'interface* d'une classe qui représente l'ensemble de ses méthodes **publiques** (encapsulation).

Interface

- Par défaut dans une interface :
  - les méthodes spécifiées sont `public abstract`
  - les constantes définies sont `public static final`

- Une classe peut ensuite implémenter une interface
  - ceci est précisé par le mot clé `implements`
  - si la classe n'implémente pas toutes les méthodes de l'interface, elle doit être déclarée `abstract`

Interface

- Une classe peut :
  - implémenter plusieurs interfaces par implements
    Ex.: **class Toto implements Cloneable, Iterable**
  - mais n'hérite que d'une seule classe par extends

- Une interface peut hériter elle-même :
  - d'une autre interface par extends
  - mais pas d'une classe normale

- Ce mécanisme permet de traduire un héritage multiple

Implémentation d'une interface

La définition d'une interface, o n y utilise simplement le mot-clé ***interface*** à la place de *class*

```
public interface I
{ void f(int n) ; // en-tete d'une methode f (public abstract facultatifs)
void g() ;        // en-tete d'une methode g (public abstract facultatifs)
}
```

Implémentation d'une interface

Lorsqu'on définit une classe, on peut préciser qu'elle implémente une interface donnée en utilisant le mot-clé ***implements***, comme dans :

```
class A implements I
{   // A doit (re)definir les methodes f et g prevues dans l'interface I
}
```

Une même classe peut implémenter plusieurs interfaces

Variables de type interface et polymorphisme

on peut définir des variables de type interface :

```
public interface I { .....}
.....
I i ; // i est une référence a un objet d'une classe implementant l'interface I
```

- Bien entendu, on ne pourra pas affecter à *i* une référence à quelque chose de type *I* puisqu'on ne peut pas instancier une interface,

En revanche, on pourra affecter à *i* n'importe quelle référence à un objet d'une classe implémentant l'interface *I* :

```
class A implements I { ..... }
.....
I i = new A(...) ; // OK
```

Exemple d'utilisation de variables de type interface

```
interface Affichable
{ void affiche() ;
}
class Entier implements Affichable
{ public Entier (int n)
{ valeur = n ;
}
public void affiche()
{ System.out.println ("Je suis un entier de
valeur " + valeur) ;
}
private int valeur ;
}
class Flottant implements Affichable
{ public Flottant (float x)
{ valeur = x ;
}
public void affiche()
{ System.out.println ("Je suis un flottant de
valeur " + valeur) ;
}
private float valeur ;
}
```

```
public class Tabhet4
{ public static void main (String[] args)
{ Affichable [] tab ;
tab = new Affichable [3] ;
tab [0] = new Entier (25) ;
tab [1] = new Flottant (1.25f) ; ;
tab [2] = new Entier (42) ;
int i ;
for (i=0 ; i<3 ; i++)
tab[i].affiche() ;
}
}
```

```
Je suis un entier de valeur 25
Je suis un flottant de valeur 1.25
Je suis un entier de valeur 42
```

Interface et classe dérivée

une classe dérivée peut implémenter une interface (ou plusieurs) :

```
interface I
{ void f(int n) ;
void g() ;
}
class A { ..... }
class B extends A implements I
{ // les mmethodes f et g doivent soit etre deja definies dans A,
// soit definies dans B
}
```

On peut même rencontrer cette situation :

```
interface I1 { ..... }
interface I2 { ..... }
class A implements I1 { ..... }
class B extends A implements I2 { ..... }
```

Différences entre interface et héritage

- Une interface fournit simplement un contrat à respecter sous forme d'en-têtes de méthodes.

- La classe implémentant l'interface est responsable de leur implémentation.

- Des classes différentes peuvent implémenter différemment une même interface, alors que des classes dérivées d'une même classe de base en partagent la même implémentation.

- On dit souvent que Java ne dispose pas de l'héritage multiple mais que ce dernier peut être avantageusement remplacé par l'utilisation d'interfaces

- En effet, une classe implémentant plusieurs interfaces doit fournir du code (et le tester !) pour l'implémentation des méthodes correspondantes.

- Les interfaces multiples assurent donc une aide manifeste à la conception en assurant le respect du contrat.

- En revanche, elles ne simplifient pas le développement du code, comme le permet l'héritage multiple.

Les classes anonymes

Java permet de définir ponctuellement une classe, sans lui donner de nom. Cette particularité a été introduite par la version 1.1 pour faciliter la gestion des événements.

Exemple de classe anonyme

Il est possible de créer un objet d'une classe dérivée de *A*, en utilisant une syntaxe de cette forme :

```
A a ;
a = new A() { // champs et methodes qu'on introduit dans
                // la classe anonyme derivee de A
} ;
```

Tout se passe comme si l'on avait procédé ainsi :

```
A a ;
class A1 extends A { // champs et methodes specifiques a A1
} ;
.....
a = new A1() ;
```

Les classes anonymes

Exemple d'utilisation d'une classe anonyme dérivée d'une autre

```
class A
{ public void affiche() { System.out.println ("Je suis un A") ;
}
}
public class Anonym1
{ public static void main (String[] args)
{ A a ;
a = new A() { public void affiche ()
{ System.out.println ("Je suis un anonyme derive de A") ;
}
} ;
a.affiche() ;
}
}
```

Je suis un anonyme derive de A

Notez bien que si *A* n'avait pas comporté de méthode *affiche*, l'appel *a.affiche()* aurait été incorrect, compte tenu du type de la référence *a* (revoyez éventuellement les règles relatives au polymorphisme). Cela montre qu'une classe anonyme ne peut pas introduire de nouvelles méthodes

Les classes anonymes

Exemple d'utilisation d'une classe anonyme implémentant une interface

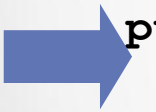
```
interface Affichable
{ public void affiche() ;
}
public class Anonym2
{ public static void main (String[] args)
{ Affichable a ;
a = new Affichable()
{ public void affiche ()
{ System.out.println ("Je suis un anonyme implementant Affichable") ;
}
} ;
a.affiche() ;
}
}
```

Je suis un anonyme implementant Affichable

Notez bien que si *A* n'avait pas comporté de méthode *affiche*, l'appel *a.affiche()* aurait été incorrect, compte tenu du type de la référence *a* (revoyez éventuellement les règles relatives au polymorphisme). Cela montre qu'une classe anonyme ne peut pas introduire de nouvelles méthodes

Exemple : interfaces Itérateurs (1/5)

```
// spécification des interfaces itérateurs
interface Iterateur {
    public abstract Object suivant();
    public abstract void debut();
    public abstract boolean fin();
}
interface IterateurInverse {
    public abstract Object suivantInverse();
    public abstract void debutInverse();
    public abstract boolean finInverse();
}
```



```
public class Tableau extends Object implements Iterateur,
  IterateurInverse {

    private Object[ ] tabElements; // le tableau d'objets
    private int nbElem;  // nombre d'objets dans le tableau
    private int courant;//indice de l'élt courant dans le parcours
```

Exemple (2/5)

```
  // constructeur
Tableau(int taille) {
        tabElements = new Object [taille];
        courant = 0;
        nbElem = 0;
}
//ajout d'un element
public void ajouterElement(Object elem) {
        if (nbElem<tabElements.length)
                tabElements[nbElem++]=elem;
}
// pour savoir si le tableau est vide
public boolean estVide() { return (nbElem == 0); };
```

Création d'une référence (**`tabElements`**) vers un tableau de références d'Object

Exemple (3/5)

```
// implémentation de l'interface Iterateur
public Object suivant() {
    if (nbElem>=0 && courant<nbElem) ;
        return tabElements[courant++];
}
public void debut() { courant = 0;}
public boolean fin() { return (courant >= nbElem);}

 // implémentation de l'interface Iterateur Inverse
 public Object suivantInverse() {
    if (nbElem>=0 && courant<nbElem) ;
        return tabElements[courant--];
}
public void debutInverse() { courant = nbElem-1;}
public boolean finInverse() { return (courant < 0);}
} // fin de la classe Tableau
```

Exemple (4/5)

```
class TestInterface {
  public static void main(String[] argv) {

    Tableau t = new Tableau(5);
    Random r = new Random(); // pour le remplissage aléatoire

     for (int i = 0; i < 5; i++) {
       Integer j = new Integer(Math.abs(r.nextInt()%51));
       t.ajouterElement(j);
     }
```

Exemple (5/5)

```
// affichage par les itérateurs

    System.out.println("\nParcours par l'iterateur\n");
    for (t.debut(); ! t.fin(); )
        System.out.println(t.suivant());

    System.out.println("\nParcours par l'iterateur inverse");
    for (t.debutInverse(); ! t.finInverse(); )

      System.out.println(t.suivantInverse());
  }

} // fin de la classe TestInterface
```