UEF4.3.Programmation Orientée Objet

Mme SADEG et Mme Bousbia
s_sadeg@esi.dz, n_bousbia@esi.dz
Ecole nationale Supérieure d'Informatique (ESI)

Classes abstraites et interfaces

Problème 1

- On souhaite disposer d'une hiérarchie de classes permettant de manipuler des figures géométriques. On veut qu'il soit toujours possible d'étendre la hiérarchie en dérivant de nouvelles classes mais on souhaite pouvoir imposer que ces dernières disposent toujours des méthodes suivantes :
  - void calculerSuperficie ()
  - void calculerPerimetre ()
  - Void afficher()

- Quelle solution proposez-vous?

Les classes abstraites

Une classe abstraite est une classe qui ne peut pas être instanciée. Elle ne peut servir que de classe de base pour être dérivée.

```
abstract class A {……}
```

Dans une classe abstraite on trouve des méthodes et des attributs mais aussi des méthodes dites abstraites.

C'est-à-dire, des méthodes dont on fournit uniquement la signature et le type de la valeur de retour

```
abstract class A
{ public void f(){….}
  public abstract void g (int n);
}
```

Les classes abstraites

Quelques règles

1. Une classe qui comporte une ou plusieurs méthodes abstraites est abstraite. Et de ce fait, **elle ne sera pas instanciable**
2. Une méthode abstraite **ne peut pas être privée** (puisqu'elle est destinée à être redéfinie dans une classe dérivée)
3. Une classe dérivée d'une classe abstraite n'est pas obligée de redéfinir toutes les méthodes abstraites de sa classe de base et peut même n'en redéfinir aucune et restera abstraite elle-même
4. Une classe dérivée d'une classe non abstraite peut être déclarée abstraite et / ou contenir des méthodes abstraites

Solution au problème

- Il suffit de créer une classe abstraite Figure disposant des en-têtes des méthodes qu'on souhaite voir redéfinies dans les classes dérivées, en leur associant le mot clé *abstract :*

```
abstract class Figure {
abstract public void afficher() ;
abstract public void calculerSuperficie();
abstract public void calculerPerimetre();
...
}
```

- Les classes decendantes de la hiérarchie de figures seront alors simplement définies comme classes dérivées de *Figure et elles devront définir les méthodes ,* par exemple :

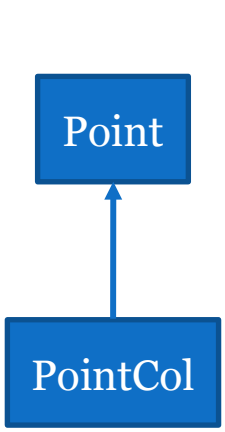
```
class Triangle extends Figure {
public void afficher() {.....}
public void calculerSuperficie(){....}
public void calculerPerimetre(){....}
}
```

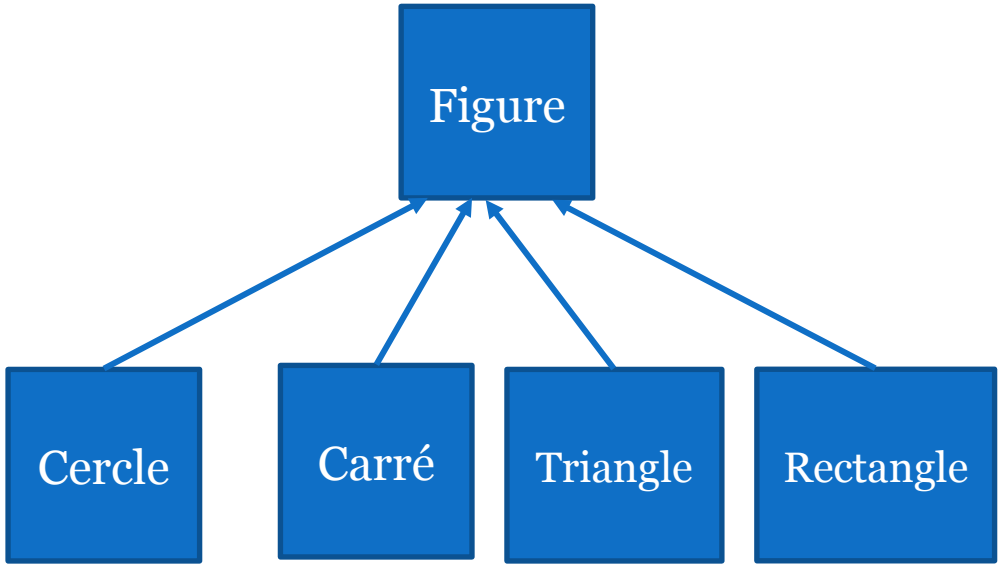
Intérêt des classes abstraites

- Les classes abstraites facilitent la conception orientée objet car elles permettent de placer dans une classe toutes les fonctionnalités dont doivent disposer toutes ses descendantes:
  - Soit sous forme complète de méthodes (non abstraites) et de champs communs à toutes les classes descendantes
  - Soit sous forme d'interface de méthodes abstraites dont on est sûr qu'elles existeront dans toute classe dérivée

Problème 2

- Considérons la hiérarchie des classes des figures géométriques et celle des points. Supposons que pour les besoins d'une application on souhaite imposer le fait que seuls les cercles et les points colorés soient déplaçables sur le plan. Que proposez vous comme solutions ?





Les interfaces

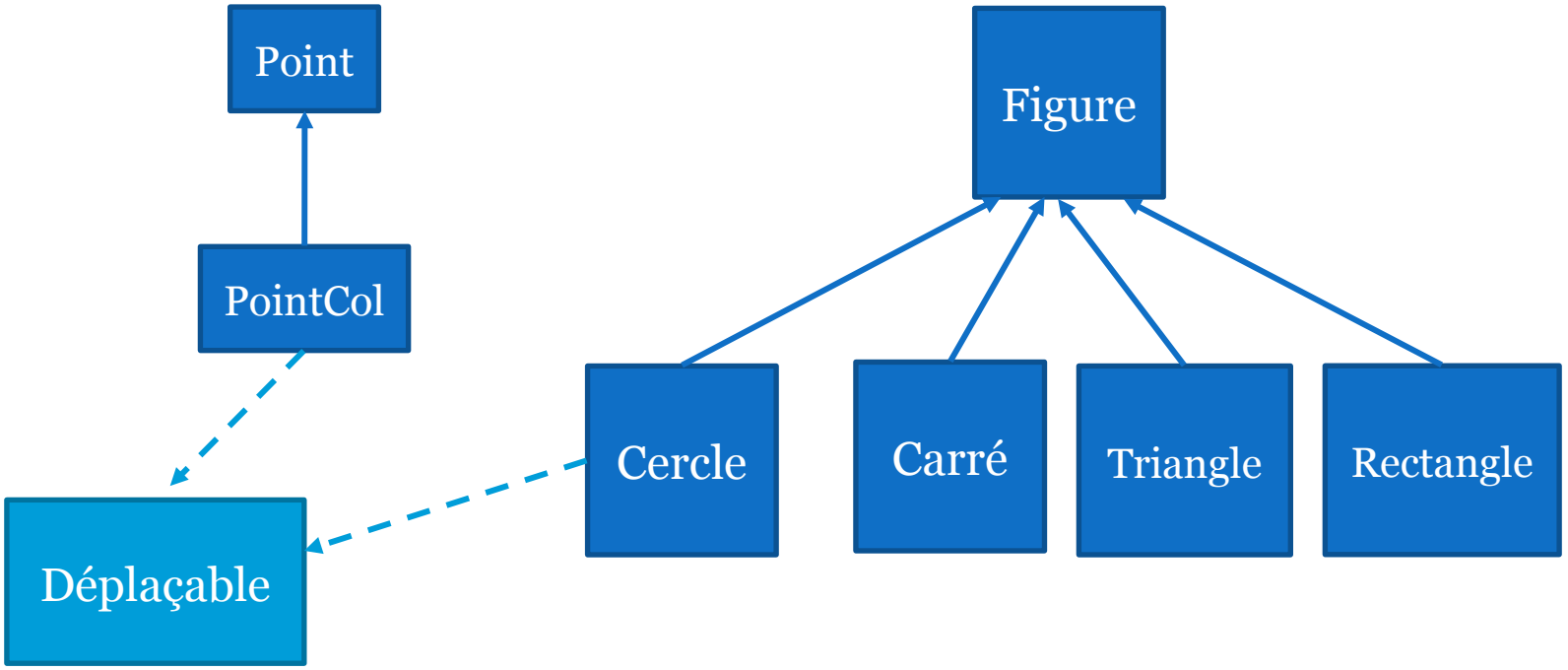
- Si on considère une classe abstraite n'implantant aucune méthode et aucun champ (hormis des constantes), on aboutit à la notion d'***interface***

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

- Si une classe **implémente** une interface, elle **s'engage** à fournir un code à ses méthodes (même vide)

Solution a problème

- Une solution serait donc de créer une interface que l'on nommera ***Deplaçable*** contenant la méthode abstraite ***deplacer (int dx, int dy)*** que seules les classes PointCol et Cercle implémenteront. Elle devront de ce fait donner un corps (redéfinir) la méthode deplacer (int dx, int dy) .



Les interfaces

Définition d'une interface

On définit une interface comme on définit une classe en remplaçant le mot clé **class** par **interface**

```
public interface I{
final int MAX = 100;
void f(int n);// public et abstract sont facultatifs
void g();
}
```

Les méthodes d'une interface sont toutes abstraites et publiques, il n'est donc pas obligatoire de le mentionner

Les interfaces

Implémentation d'une interface

Une classe peut implémenter une interface

```
class A implements I
{// A doit redéfinir les méthodes de I sinon erreur à la compilation}
```

Une classe peut implémenter plusieurs interfaces

```
public interface I1 {
 void f();
}
public interface I2{
 void g()
}
class A implements I1,I2
{// A doit définir les méthodes f et g
}
```

Variables de type interface et polymorphisme

Il est possible de définir des variables de type interface

```
public intarface I { …. }
…
I i;   //i est une référence à un objet d'une classe
     //implémentant l'interface I

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

Dérivation d'une interface

On peut définir une interface comme étant dérivée d'une autre interface (mais pas d'une classe) en utilisant le mot clé *extends*.

```
interface I1
{
static final MAXI = 100;
void f (int n);
}
interface I2 extends I1
{
static final MINI = 10;
void g();
}
```

```
interface I2 {
static final MINI = 10;
static final MAXI = 100;
void f (int n);
void g();
}
```

Dérivation d'une interface

Une interface peut dériver de plusieurs interfaces.
L'héritage multiple est autorisé pour les interfaces

```
interface I1 {
void f();
}
interface I2 {
void f1();
}
interface I3 extends I1,I2 {
void f2();
}
```

Conflits de noms

Exemple 1

```
interface I1{
  void f (int n);
  void g();
}

interface I2{
 void f(float x);
 void g();
}

class A implements I1,I2{
/* Pour satisfaire I1 et I2, A doit définir
 deux méthodes f: void f(int) et void f(float),
 mais une seule méthode g: void g()*/
}
```

Conflits de noms

Exemple 2

```
interface I1{
 void f (int n);
 void g();
}
interface I2{
 void f(float x);
 int g();
}
class A implements I1,I2{
/* Pour satisfaire I1 et I2, A doit contenir à la
 fois une méthode void g() et une méthode int
 g(), ce qui est impossible d'après les règles de
 surdéfinition. Une classe ne peut implémenter
 les interfaces I1 et I2 telles qu'elles sont
 définies dans cet exemple*/
}
```

Méthodes « default »

- Avant java 8 une interface ne pouvait contenir que des entêtes de méthodes et ne fournissait pas l'implémentation de ces méthodes. Chaque classe implémentant cette interface s'engageait à fournir sa propre implémentation.

- Depuis java 8, il est permis de fournir une implémentation aux méthodes dans les interfaces.

Méthodes « default »
syntaxe

```
public interface I {

/* Pas d'implémentation */

public void m1();

/* Implémentation par défaut*/

default void m2() {
// corps de la méthode
}

}
```

Méthodes « default »
utilité

- Les méthodes par défaut permettent de faire évoluer une interface sans que cela pose problème au niveau des classes qui l'implémentent.

  Exemple: cas des interfaces des API java.

- Les classes qui partagent la même implémentation d'une méthode d'une interface ne sont plus obligées de répéter le même code chacune à son niveau.

Méthodes « default »
possibilités d'utilisation

- Une classe implémentant une interface comportant une méthode par défaut (default) peut:
  - Utiliser directement cette méthode
  - Redéfinir cette méthode
- Une interface I2 dérivant d'une interface I1 peut:
  - Hériter de l'implémentation de la méthode par défaut
  - Redéfinir une méthode par défaut
  - La déclarer à nouveau, ce qui la rend abstraite

Méthodes « default »
règles d'utilisation

- Si deux interfaces I1 et I2 déclarent la même méthode m () mais proposent des implémentations incompatibles, que se passe-t-il pour la classe implémentant ces deux interfaces?

- La classe doit redéfinir la méthode pour résoudre le conflit.

- Pour appeler la méthode par défaut d'une des deux interfaces, une syntaxe particulière a été ajoutée:
  I2.**super**.m();

Méthode default

Exemple 1

```
interface I1{
 default void m(int n) {…..}

  void g();
}

interface I2{
default void m(int n) {…..}
 void f(float x);
 void g();
}

class A implements I1,I2{
/* Pour satisfaire I1 et I2, A doit les méthodes de I1 et
 I1, et redéfinir void m(int)*/
void m(int n) {
I1.super.m();
}
}
```

Méthodes statiques dans les interfaces

- C’est une autre nouveauté de java 8.
- Les interfaces peuvent contenir des méthodes statiques et elles sont utilisées exactement de la même manière que celles que peuvent contenir des classes.

Exemple de l'interface Comparable

- Une classe implémente l'interface *comaprable* si ses objets peuvent être ordonnés selon un ordre particulier.
  - Par exemple la classe String implémente Comparable parce que les chaines de caractères peuvent être ordonnées selon l'ordre alphabétiques.
- Les classes numériques comme Integer et Double implémentent Comparable parce que les nombres peuvent être ordonnés selon l'ordre numérique

Exemple de l'interface comparable

- L'interface Comparable consiste en une seule méthode (et pas de constantes) `int compareTo(T obj)` qui compare l'objet à un objet de type T .
- `A.compareTo(B)` retourne un entier négatif si l'objet A est plus petit que B, zéro si les deux objets sont égaux et un entier positif si l'objet A est plus grand que l'objet B.
- Exemple:

"Bonjour ".compareTo ("Bonsoir ") renvoie un entier négatif.

Différences entre interface et héritage

- Une interface fournit un contrat à respecter sous forme d'en-tête de méthodes auxquelles la classe implémentant l'interface s'engage à fournir un corps (qui peut être vide ou contenir des instructions).

- Des classes différentes peuvent implémenter différemment une même interface alors que les classes dérivées d'une même classe de base partagent la même implémentation des méthodes héritées

- Même si Java ne dispose pas de l'héritage multiple, ce dernier peut être remplacé par l'utilisation d'interfaces, avec l'obligation d'implémenter les méthodes correspondantes.

Exercice

Créez une classe CompteEnBanque munie d'un seul attribut solde, et définissez l'égalité de deux objets comptes en banque, de telle manière qu'elle soit vérifiée dès lors que les deux soldes sont égaux.

Solution

```
class CompteEnBanque implements Comparable{

    private float solde;

    public float getSolde(){ return solde;}
    public void setSolde(float s){ solde=s;}

    public int compareTo(Object cmp) {

    float solde2 = ((CompteEnBanque)cmp).getSolde();

  if(this.solde>solde2)
     return 1;
    else if(this.solde <solde2)
            return -1;
    else
            return 0;
}
}
```

Solution (une autre version)

```
class CompteEnBanque implements Comparable<CompteEnBanque>{

    private float solde;

    public float getSolde(){ return solde;}
    public void setSolde(float s){ solde=s;}

    public int compareTo(CompteEnBanque cmp) {

    float solde2 = cmp.getSolde();

  if(this.solde>solde2)
     return 1;
    else if(this.solde <solde2)
            return -1;
    else
            return 0;
}
}
```