Remise à niveau Programmation

Programmation Objet avec Java

Classes Abstraites et interfaces

#5

Licence Pro
Métiers de l'Informatique : Conception, Développement,
Test de Logiciels, Parcours "Génie Logiciel, Système d'Information"
2020-2021

Classes abstraites

Contrat

Une classe définit un contrat : l'ensemble des services qu'elle s'engage à rendre.
Chaque méthode remplit une part du contrat :

- la signature (et la documentation) de la méthode décrit le service rendu. C'est une spécification qui répond à la question : quoi ?
- les instructions de la méthode établissent la façon dont ce service est rendu. C'est un service concret (implémentation) qu'on peut exécuter et qui répond à la question : comment ?

Contrat de classe et méthode abstraite

```
public abstract class Piece {                          // Classe Abstraite
  private int ligne ;                                  // definissant une
  private int colonne ;                                // une pièce sur un
                                                       // damier
  public Piece( int li, int co)
  {
     this.ligne = li ;
     this.colonne = co ;
  }

  // vérifie que le coup arrivant sur la
  // case (li, co) est valide
  public abstract boolean coupOk(int li, int co) ; // Méthode Abstraite
}
```

Méthode abstraite

Une méthode abstraite ne possède pas d'instructions (d'implémentation).
C'est une signature définissant le service que doit rendre la méthode, sans préciser la façon dont elle le rendra. Elle définit donc une spécification (service abstrait)

- Le mot clef **`abstract`** de la déclaration de la méthode **`coupOK()`** indique que cette méthode est une méthode abstraite. Aucune implémentation (corps d'instructions) ne lui est donc associée.

Classe abstraite

Toute classe comportant au moins une méthode abstraite est nécessairement abstraite.
Une classe peut être déclarée abstraite même si elle ne comporte aucune méthode abstraite.

- Le mot clef **`abstract`** de la déclaration de la classe **`Piece`**, indique que la classe est abstraite.

Une classe abstraite ne peut être instanciée (il ne peut pas y avoir d'objet de cette classe).

```
{
     Piece p = new Piece() ;       // REJET DU COMPILATEUR ! ! ! !
}
```

Abstraites/Concrètes

Méthodes/Classes abstraites

Une classe *abstraite* sert de modèle pour créer des classes *concrètes* (non abstraites) qui en hériteront.

```
public abstract class Pièce {
  // ------------------------------------------------------------ <
  // Une classe abstraite définit une implémentation partielle    <
  // (les méthodes concrètes ont des instructions, les méthodes   <
  // abstraites n'en ont pas)                                     <
  private int ligne ;              //                           <
  private int colonne ;            //                           <
                                   //                           <
  public Pièce( int li, int co) {  //                           <
     this.ligne = li ;             //                           <
     this.colonne = co ;           //                           <
  }  // ------------------------------------------------------- <

  public abstract boolean coupOk(int l, int c) ;
}
```

Remarque

Une classe abstraite peut comporter des méthodes concrètes. Il s'agit alors d'une implémentation partielle dont pourront hériter les classes filles. Ces dernières pourront éventuellement redéfinir ces classes concrètes si il le faut.

Méthodes/Classes concrètes

Les classes concrètes qui hériteront d'une classe abstraite devront compléter l'implémentation en fournissant des méthodes concrètes pour **TOUTES** les méthodes abstraites

```
public class Tour extends Pièce {
  public Tour( int li, int co){ super(li, co) ; }

  public boolean coupOk(int li, int co){    // Implémentation concrète
     return ( this.ligne == li || this.colonne == co ) ;
  };
}
```

```
public class Fou extends Pièce {
  public Fou( int li, int co){ super(li, co) ; }

  public boolean coupOk(int li, int co) {   // Implémentation concrète
     return ( Math.abs( this.ligne–li) == Math.abs( this.colonne-co)) ;
  };
}
```

Remarque

Une classe concrète propose une implémentation pour tous les services, elle peut donc être instanciée.

Liaison Dynamique/Statique

Les mécanismes à l'œuvre dans une hiérarchie de classe dans laquelle existe un polymorphisme d'héritage sont toujours valide ici.

Soit la classe de test suivante :

```
public class TestPièce {
  public static void main( String [] args)
  {
     Pièce [] lesPièces = { new Tour( 4, 2), new Fou( 5, 6), ...} ;

     for (int i = 0 ; i < lesPièces.length ; i++)
     {
          ...
          int li = nextInt() ;          // Saisie de la nouvelle
          int co = nextInt() ;          // position pour la pièce

          if ( ! lesPièces[i].coupOK( li, co) )
               System.out.println("Déplacement interdit") ;
          else
               System.out.println("Déplacement autorisé") ;;
     }
  }
}
```

Liaison Statique :

Le compilateur teste si une méthode de signature **`Pièce::coupOK( int, int)`** est définie (dans la classe **`Pièce`**).
Il en trouve une ! Le fait qu'elle soit abstraite ne pose pas de problème.

Liaison Dynamique :

La méthode polymorphe la plus spécifique de l'objet référencé est invoquée. Comme l'objet référencé est une instance elle vient forcément d'une classe concrète qui a implémenté la classe abstraite.

- pour i = 0 c'est la méthode **`coupOK( int, int)`** de la classe **`Tour`** qui est invoquée ;
- pour i = 1 c'est la méthode **`coupOK( int, int)`** de la classe **`Fou`** qui est invoquée ;

Remarque

La liaison statique vérifie sur la classe abstraite que le service existe, et la liaison dynamique invoque le service implémenté.

Interfaces - Intro

Rappel Classe Concrète/Classe abstraite :

Concrètes : Elles définissent un contrat et des services concrets. Elles ne contiennent que des méthodes qui ont une implémentation concrète (par des instructions)
Abstraites : Elles contiennent des méthodes concrètes et abstraites. Elles définissent un contrat mixte avec des méthodes concrètes définissant des services concrets (avec implémentation) et des méthodes abstraites définissant une spécification (service abstrait) sans implémentation

Interfaces définissent un contrat purement abstrait

Elles ne contiennent que :

- des méthodes abstraites publiques
- des constantes de classe publiques

Remarque

Une interface répond à la question « *quoi ?* » et ne répond pas à la question « *comment ?* ». Elle ne définit aucune implémentation.

Exemple

```
public interface FigureFermée {

    // couleur de remplissage par défaut
    public static final String DFLT_COULEUR_IN = "Blanc" ;

    abstract double périmètre() ;
    abstract double aire() ;
    ...
}
```

Une interface n'est pas une classe. Sa déclaration utilise le mot clef **`interface`** et non **`class`**.

Remarque

- Il est possible d'omettre **`public static final`** pour les variables car cette déclaration est implicite dans une interface.
- Il est possible d'omettre **`abstract`** pour les méthodes cat cette déclaration est implicite dans une interface.

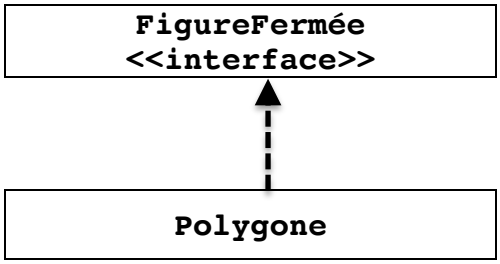
Remarque

Une interface a une vocation déclarative. C'est un modèle de services qui sont réalisés par d'autres classes. L'implémentation est déléguée aux classes qui implémenteront l'interface (*ie* le contrat)

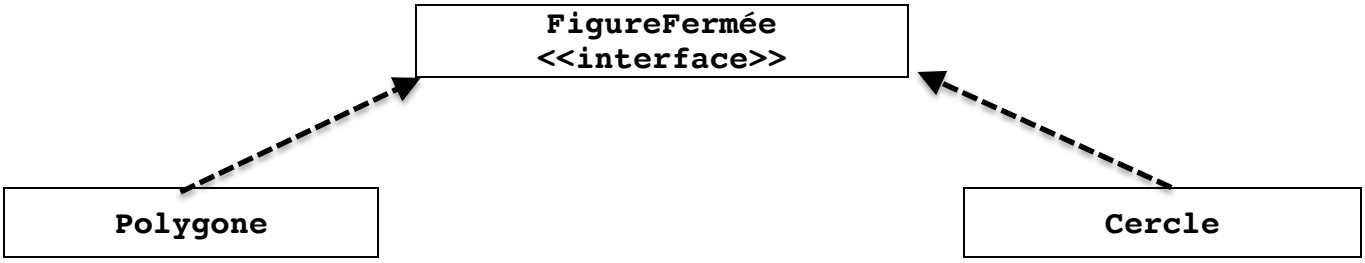
Interfaces - Implémentation

Une interface est un modèle abstrait :

- Le contrat abstrait qu'elle définit (les spécifications) doit être implémenté par une classe pour être utilisé
- Une classe implémente une interface lorsqu'elle donne une implémentation (un corps d'instructions) aux méthodes abstraites définies dans l'interface



Le trait discontinu signifie que la classe **`Polygone`** implémente l'interface **`FigureFermée`** (ne pas confondre avec l'héritage).



Remarque

Une interface peut (c'est le cas le plus courant) avoir plusieurs implémentations. Chaque implémentation est une façon différente de réaliser le contrat abstrait (les spécifications) défini par l'interface.

Autrement dit, les corps d'instructions associés aux méthodes concrètes des classes concrètes qui implémentent le contrat abstrait de l'interface peut (doit ?) être différents d'une classe concrète à l'autre. Les classes concrètes rendent le service promis par l'interface, mais de façon différente, chacune à sa manière avec son propre code.

Remarque

La classe concrète qui implémente une interface le déclare dans son entête au moyen du mot clef **`implements`** suivit du nom de l'interface implémentée.

Interfaces - Implémentation

```
public class Cercle implements FigureFermée
{
     private Point centre;
     private double rayon;
     public Cercle(Point centre, double rayon) { ... }
     public Cercle(Cercle c) { ... }
     ...
     public double périmètre(){          // IMPLÉMENTATION CONCRÈTE
          return ( 2 * Math.PI * this.rayon ) ;
     }

     public double aire(){               // IMPLÉMENTATION CONCRÈTE
          return ( Math.PI * this.rayon * this.rayon) ;
     }
}
```

```
public class Polygone implements FigureFermée
{
     private PointPlan [] sommets ;
     public Polygone (PointPlan [] som) { ... }
     public Polygone( Polygone p ){ ... }
     public int nbDeSommets(){ ... }
     public PointPlan getSommet( int i){ ... }
     ...
     public double périmètre(){          // IMPLÉMENTATION CONCRÈTE
          Point p0 ;
          Point p1 ;
          double per = 0 ;
          for (int i = 0 ; i < this.nbDeSommets() ; i++ )
          {
               p0 = this.getSommet(i) ;
               if ( i+1 < this.nbDeSommets())
                    p1 = this.getSommet(i+1) ;
               else
                    p1 = this.getSommet(0) ;
               double dx =    p1.getAbscisse() – p0.getAbscisse() ;
               double dx =    p1.getAbscisse() – p0.getAbscisse() ;
               per += Math.sqrt( dx * dx + dy * dy )
          }
          return per ;
     }

     public void aire(){                 // IMPLÉMENTATION CONCRÈTE
          ... ;
     }
}
```

Interfaces - Implémentation

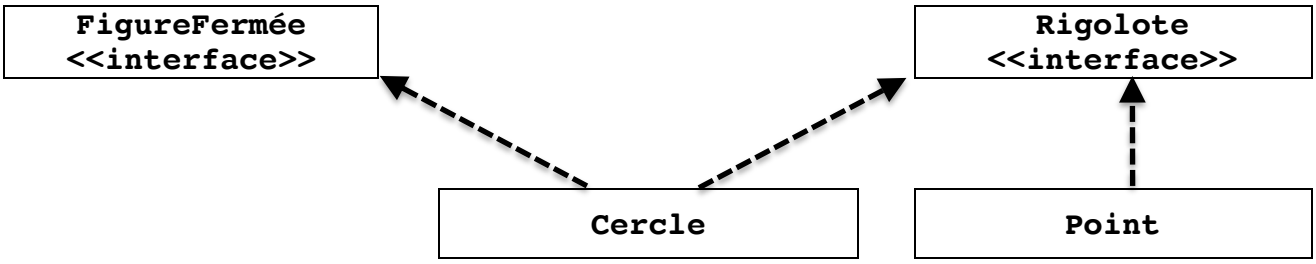
Exemple d'utilisation

Les signatures des méthodes de **`Cercle`** et **`Polygone`** qui dérivent de l'implémentation de l'interface **`FigureFermée`** sont identiques : ces 2 classes s'utilisent donc strictement de la même façon pour cette partie.

```
public class TestFigureFermée
{
   public static void main( String [] args)
   {
     Cercle cercle = new Cercle( new PointPlan(0., 0.), 12.) ;
     Polygone carré = new Polygone(     {
                                        new PointPlan( 0., 0.),
                                        new PointPlan( 0., 1.),
                                        new PointPlan( 1., 1.),
                                        new PointPlan( 1., 0.)
                                        } ) ;

   //     Appels/signature identiques          VVVVVVVVVVVVVVVVV

     System.out.println( "Aire cercle = ",        cercle.aire()) ;
     System.out.println( "Périmètre cercle = ",  cercle.périmètre ()) ;

     System.out.println( "Aire carré = ",        carré.aire()) ;
     System.out.println( "Périmètre carré = ",   carré.périmètre ()) ;
   }
}
```

Plusieurs interfaces

Si l'héritage multiple n'est pas permis en Java, il est par contre parfaitement possible de « faire » implémenter plusieurs interfaces à une même classe :



La classe **`Point`** n'implémente qu'une seule interface alors que la classe **`Cercle`** en implémente deux.
En conséquence,

- La classe **`Point`** doit proposer une implémentation concrète pour les seuls services de l'interface **`Rigolote`** alors que
- la classe **`Cercle`** doit proposer une implémentation concrète de l'ensemble des services déclarés par l'interface **`FigureFermée`** et par l'interface **`Rigolote`**.

```
public interface Rigolote
{
     public String blague() ;
}
```

```
public interface FigureFermée {

    // couleur de remplissage par défaut
    public static final String DFLT_COULEUR_IN = "Blanc" ;

    abstract double périmètre() ;
    abstract double aire() ;
    ...
}
```

Dans le cas où une classe implémente 2 (ou plus) interfaces, elle le spécifie dans sa déclaration en indiquant par le mot-clef **`implements`**, l'ensemble des interfaces implémentées en séparant leurs noms par une virgule :

```
public class Cercle implements FigureFermée, Rigolote { ... }
```

Plusieurs interfaces

```
public class Cercle implements FigureFermée, Rigolote
{
     private Point centre;
     private double rayon;
     public Cercle(Point centre, double rayon) { ... }
     public Cercle(Cercle c) { ... }
     ...

     // ------------------------------------------------------
     // IMPLÉMENTATION CONCRÈTE DE FigureFermée

     public double périmètre(){
          return ( 2 * Math.PI * this.rayon ) ;
     }
     public double aire(){
          return ( Math.PI * this.rayon * this.rayon) ;
     }

     // ------------------------------------------------------
     // IMPLÉMENTATION CONCRÈTE DE Rigolote

     public double blague(){            // IMPLÉMENTATION CONCRÈTE
          System.out.println("On demande à un ingénieur, un physicien et
un mathématicien de construire un enclos pour y mettre des
moutons.\nL'ingénieur regroupe les moutons et construit une clôture
autour.\nLe physicien construit une palissade en ligne droite qui est en
réalité un cercle de diamètre infini.\nLe mathématicien construit une
petite clôture autour de lui et se définit comme l'extérieur.") ;
     }
}
```

```
public class Point implements Rigolote
{
     private double abscisse;
     private double ordonnée;
     public Point(double x, double y) { ... }
     public Point(Point centre) { ... }

     // ------------------------------------------------------
     // IMPLÉMENTATION CONCRÈTE DE Rigolote

     public double blague(){       // IMPLÉMENTATION CONCRÈTE
          System.out.println " Qu'est-ce qu'un ours polaire ?\nUn ours
cartésien après un changement de coordonnées.\n") ;
     }
}
```

Une interface définit un type

Une classe définit un type concret,
Une classe abstraite ou une interface (classe abstraite particulière) définit un type abstrait.

On peut utiliser ce type pour référencer un objet mais pas pour créer un objet (**new** interdit)
Le type d'une interface est plus général que le type de ses implémentations : on peut donc utiliser le type défini par une interface à la place du type défini par une implémentation de cette interface

Exemple

Un **`Cercle`** ou un **`Polygone`** peuvent être référencés comme une **`FigureFermée`**.

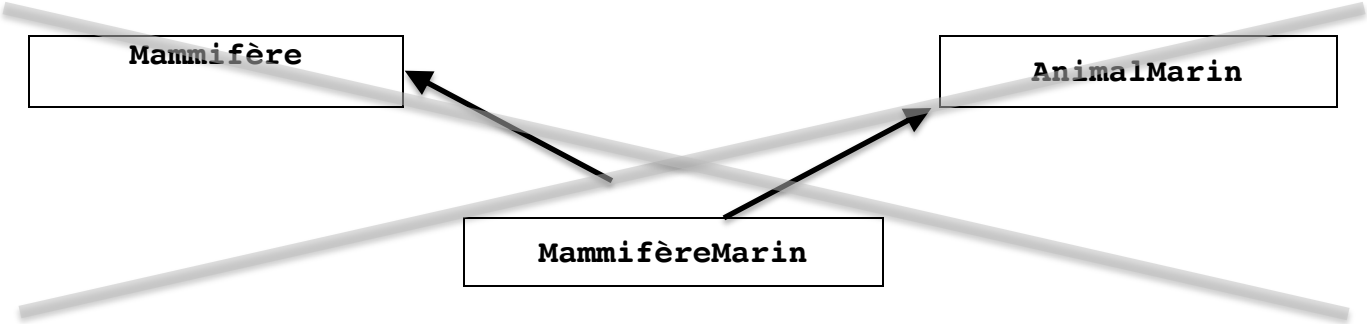
```
public static void main( String [] args)
{
     FigureFermée f1 = new Cercle( new PointPlan(0., 0.), 12.) ;
     FigureFermée f2 = new Polygone(    {
                                        new PointPlan( 0., 0.),
                                        new PointPlan( 0., 1.),
                                        new PointPlan( 1., 1.)
                                        } ) ;
     System.out.println( "Aire f1 = ",       f1.aire()) ;

     System.out.println( "Aire f2 = ",       f2.aire()) ;
   }
}
```

On peut placer un **`Cercle`** ou un **`Polygone`** partout ou est attendue une **`FigureFermée`**.

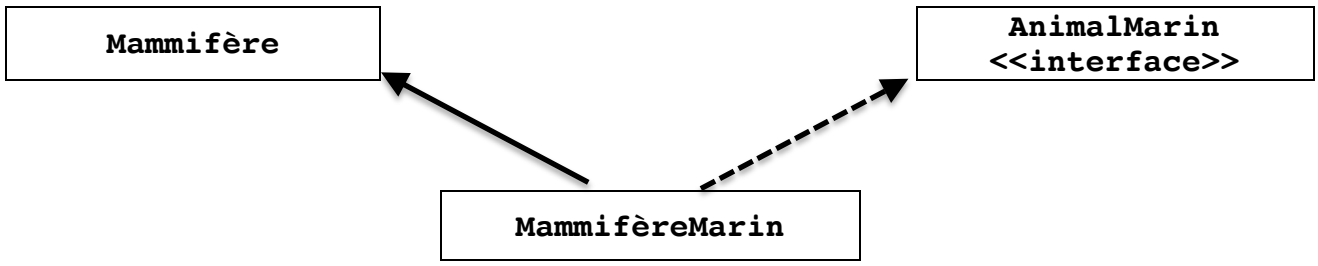
```
public class Test
{
   public static double sommeAires( FigureFermée f1, FigureFermée f2)
   {
     return f1.aire() + f2.aire() ;
   }

   public static void main( String [] args)
   {
     double somme = Test.sommeAires(
                              new Cercle( new PointPlan(0., 0.), 3.),
                              new Cercle( new PointPlan(0., 0.), 2.)
                    )
   }
}
```

Une interface définit un type

On rappelle que l'héritage multiple n'est pas possible en Java :



Les interfaces permettent de palier partiellement l'absence d'héritage multiple en java.
La classe **MammifèreMarin** hérite de **Mammifère** et implémente l'interface **AnimalMarin**




```
public class Mammifère{
     private int nbMamelles ;
     public int getNbMamelles() { return this.nbMammelles ; }
}
```

```
public interface AnimalMarin{
     public abstract void nager() ;
}
```

```
public class MammifereMarin extends Mammifere implements AnimalMarin {
     int dureePlongee ;

     public MammifereMarin(double poids,  int nbMamelles, int uneDuree){
          super(poids, nbMam) ; this.dureePlongee = d ;
     }

     public void nager ( ){
          System.out.println("je plonge et remonte apres " +
this.dureePlongee + "minutes")
     }
}
```

Une interface définit un type

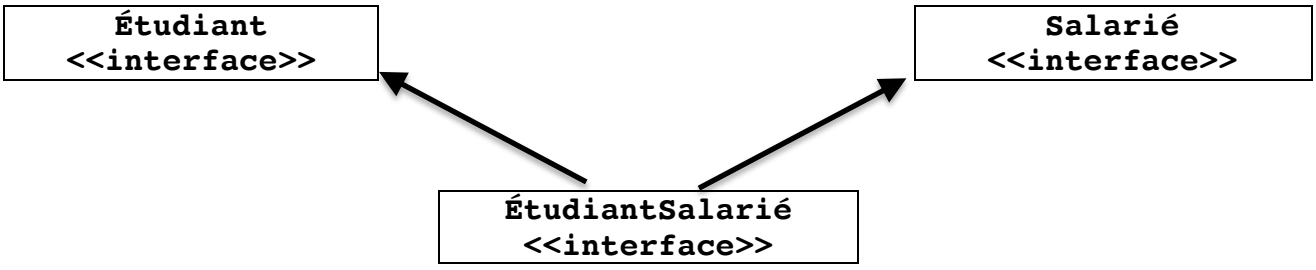
```
public class Test {
     public static void main(String [] args){

          MammifereMarin dauphin = new MammifereMarin(150, 6, 30) ;

          // Méthode héritée de la classe Mammifère
          System.out.println(dauphin.getNbMamelles( )) ;

          // Méthode de l'interface AnimalMarin implémentée
          // (concrétisée )dans la classe MammifèreMarin
          dauphin.nager( ) ;

          // Méthode prenant en paramètre un Mammifère
          SurMammiferes.surMammifere(dauphin) ;

          // Méthode prenant en paramètre un AnimalMarin
          SurAnimauxMarins.surAnimalMarin(dauphin) ;
     }
}
```

```
public class SurMammiferes {
     public static void surMammifere
(Mammifere m) { ...}
}
```

```
public class SurAnimalMarin {
     public static void surAnimalMarin
(AnimalMarin a) { ...}
}
```

Héritage entre interfaces

Une interface peut hériter d'une ou plusieurs autres interfaces : L'héritage multiple est donc possible entre interfaces !



L'interface fille (**ÉtudiantSalarié**) hérite des constantes de classe et des méthodes abstraites des interfaces mères (**Étudiant**, **Salarié**).

```
public interface EtudiantSalarie extends Etudiant, Salarie
{ ... }
```

Interface comparable

Il existe un certain nombre d'interfaces pré-définies par Java qu'il peut être intéressant d'implémenter dans les classes développées par un programmeur.

Parmi celle-ci l'interface **Comparable**.
Contient une méthode abstraite définissant un ordre entre objets

```
public interface Comparable
{
     public int compareTo(Object o) ;
}
```

Cette méthode retourne –1, 0 ou 1 selon que l'objet référencé par **this** est inférieur, égal ou supérieur à l'objet référence par **o**.

Toute classe implémentant l'interface **Comparable** définit un ordre total sur les objets qu'elle crée.

Exemple 1

```
public class Point implements Comparable
{
     private double abscisse ;

     ...
     // retourne –1, 0 ou 1 selon que le Point courant est d'abscisse
     // inférieure, égale ou supérieure à l'abscisse du Point référencé
     // par o
     public int compareTo(Object o){
          double x1 = this.abscisse ;
          double x2 = ((Point) o).abscisse ;
          if (x1 < x2) return –1 ;
          if (x1 == x2) return 0 ;
          return 1 ;
     }
}
```

Interface comparable

Exemple 2

```
public class Rectangle implements Comparable
{
     private double longueur ;
     private double largeur ;

     ...
     // retourne –1, 0 ou 1 selon que le Rectangle courant est d'aire
     // inférieure, égale ou supérieure à l'abscisse du Rectangle
     // référencé par o
     public int compareTo(Object o){
          Rectangle r = (Rectangle) o ;
          double s1= this.longueur * this.largeur ;
          double s2= r.longueur * r.largeur ;
          if (s1 < s2) return –1 ;
          if (s1 == s2) return 0 ;
          return 1 ;
     }
}
```

Utilisation de l'interface Comparable

```
public class Comparer
{
     public static boolean plusGrand(Comparable c1, Comparable c2){
          return (c1.compareTo(c2) == 1);
     }
}
```

```
{
   Point p1 = new Point(15) ;
   Point p2 = new Point(10) ;

   if (Comparer.plusGrand(p1, p2))
     System.out.println(p1 + " est plus grand que " + p2) ;

   Rectangle r1 = new Rectangle(10, 15) ;
   Rectangle r2 = new Rectangle (5, 8) ;

   if (Comparer.plusGrand(r1, r2))
     System.out.println(r1 + " est plus grand que " + r2) ;
}
```

Interface de marquage

Certaines interfaces, comme l'interface **`Cloneable`** ne contiennent rien et sont utilisées pour désigner (marquer) les classes qui les implémentent à la machine virtuelle.

Une interface de marquage est assortie d'une documentation précisant les contraintes que toute classe qui l'implémente doit satisfaire.

Exemple

Une classe **`ClasseX`** implémentant **`Cloneable`** autorise le clonage (copie) de ses instances avec la méthode **`clone()`** de la classe **`Object`**

```
{
     ClasseX x = new ClasseX();

     try
     {
          Object y = x.clone( ) ;
     }
     catch (CloneNotSupportedException e)
     {
          System.out.println(e + "clonage interdit") ;
     }

}
```

La machine virtuelle lèvera une exception si la **`ClasseX`** n'implémente pas l'interface **`Cloneable`**.

Exercices #5

Vous trouverez en téléchargement sur le lien suivant les JavaDoc et les sources des classes **`PointPlan`**, **`Polygone`**, **`Triangle`** et **`Quadrilatère`**. Les classes ont été augmentées de quelques méthodes pour faciliter la résolution de ce TD.

Question 1 :
Donnez l'entête de la classe **`Polygone`** pour signifier que cette classe réalise l'interface **`Comparable`**.

Question 2 :
Écrire dans **`Polygone`** la méthode permettant de remplir ce contrat. On considère qu'un **`Polygone`** est inférieur à un autre si son périmètre est plus petit.

Question 3 :
Dans un programme de tests :

- Définissez deux références c1 et c2 à des objets de type **`Comparable`**.
- Faites référencer à la première un nouveau **`Triangle`** avec trois points : *(2001,2000)*, *(2000, 2001)*, *(2001,2001)*
- Faites référencer à la seconde un nouveau **`Quadrilatere`** avec quatre points *(1,1)*, *(1000,1)*, *(1000,1000)*, *(1,1000)*

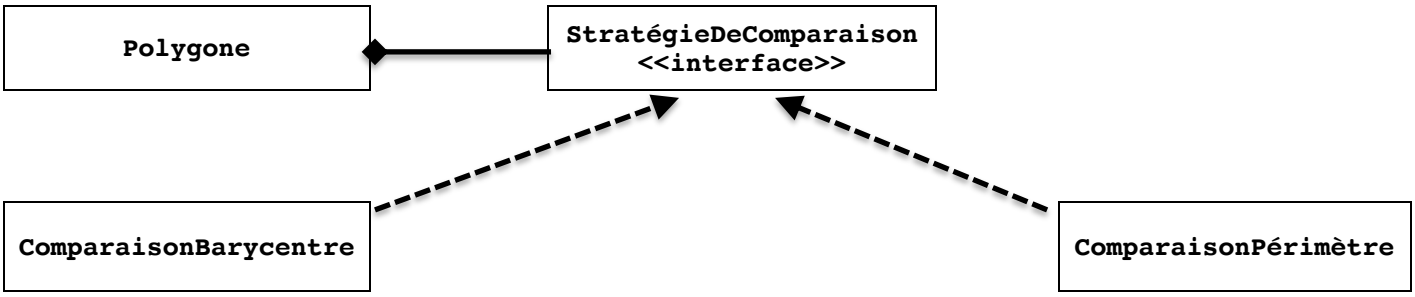
Question 4 :
Qu'affichent les instructions suivantes ?

```
{
     if (c1.compareTo(c2)==1)
          System.out.println("Supérieur strict");
     else
          System.out.println("Inférieur ou égal");
}
```

Question 5 :
Proposez une implémentation alternative de la méthode **`compareTo()`** pour que les **`Polygone`**s soient comparés selon la distance de leur barycentre à l'origine du repère orthonormé.

Nous nous proposons d'implémenter le patron de conception Strategy pour la comparaison des polygones.
Le patron de conception stratégie est utile pour des situations où il est nécessaire de permuter dynamiquement les algorithmes utilisés dans une application. Le patron stratégie est prévu pour fournir le moyen de définir une famille d'algorithmes, encapsuler chacun d'eux en tant qu'objet, et les rendre interchangeables. Ce patron laisse les algorithmes changer indépendamment des clients qui les emploient.

Ici, l'algorithme que nous souhaitons pouvoir changer est l'algorithme utilisé par la méthode **`compareTo()`** ; Nous souhaitons pouvoir passer dynamiquement, pour un jeu de polygone donné, d'une comparaison basée l'ordre défini sur les périmètre à l'ordre définit sur la distance des barycentre à l'origine du repère. La description de ce pattern est la suivante :



Question 6 :
Définissez une nouvelle interface **`StratégieDeComparaison`**. Cette interface doit proposer un contrat simple, avec une seule méthode **`comparaison(Polygone p1, Polygone p2)`**, retournant `-1`, `0` ou `1` de manière similaire à **`compareTo(.)`**

Question 7 :
Définissez une nouvelle classe **`ComparaisonBarycentre`** qui réalise l'interface **`StratégieDeComparaison`**. Inspirez-vous largement du code de **`compareTo(.)`** de la Question 5 pour écrire la méthode **`comparaison(...)`** de la nouvelle classe.

Question 8 :
Définissez une nouvelle classe **`ComparaisonPérimètre`** qui réalise l'interface **`StratégieDeComparaison`**. Inspirez-vous largement du code de **`compareTo(.)`** de la Question 5 pour écrire la méthode **`comparaison(...)`** de la nouvelle classe.

Question 9 :
Définissez dans **`Polygone`** une nouvelle variable de classe publique nommée **`comparateur`**, pour référencer une **`StratégieDeComparaison`**.

Question 10 :
Réécrivez la méthode **`compareTo(...)`** de polygone, simplement en appelant **`comparaison(...)`** de la variable de classe (statique).

Question 11 :
Proposez un programme testant le changement dynamique de stratégie de comparaison.