Université de Nice-Sophia Antipolis
POLYTECH. PeiP2

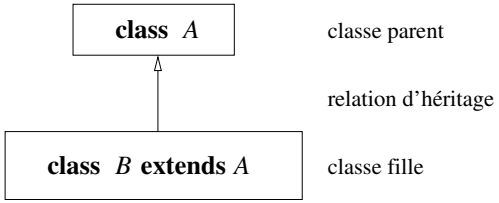
vendredi 17 novembre 2017
Durée: 2h

Algorithmique et Programamtion Java
Travaux Pratiques – Séance n° 5

---

1 Figures géométriques

En `une classe` *B* appelée la classe héritière ou fille, hérite d'une classe *A*, appelée la classe parent, en indiquant dans son en-tête le nom de la classe parent précédé du mot-clé **extends**. Le schéma ci-dessous représente la relation d'héritage entre les classes *A* et *B*. Notez le sens de la flèche qui indique le sens de l'héritage.



La classe fille *B* est héritière au sens qu'elle possède tous les attributs et les méthodes de sa classe parente *A* en plus de ses propres attributs et méthodes. Toutefois, le langage `éfinit` es règles qui peuvent modifier l'accès aux attributs et aux méthodes d'une classe. Nous avons déjà vu les accès **private** et **public**, le langage propose un accès **protected** réservé aux classes héritières [1].

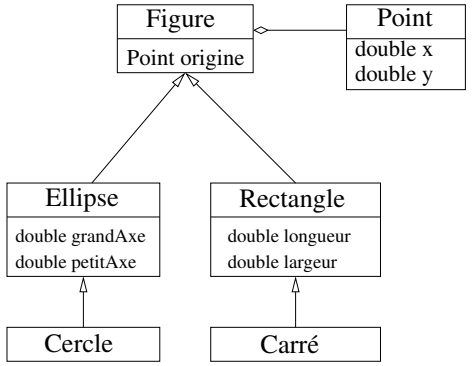
niveau	accessibilité
`private`	la classe de déclaration uniquement
`protected`	la classe de déclaration, ses héritières et le paquetage
`public`	toutes les classes

Le parent d'un objet est désigné par le mot clé **super** (on parle aussi de sa super-classe). Rappelons que **this** désigne l'objet lui-même.

Dans ce TP nous allons construire une classe `Point` représentant un point de l'espace cartésien à deux dimensions à coordonnées entières. Nous utiliserons les points pour définir des figures géométriques. La classe `Figure` représente une figure quelconque. Les figures particulières, telles des rectangles ou des ellipses seront représentées par des classes héritères de la classe `Figure`.

Le graphe ci-desous décrit les relations d'héritage et de clientèle entre les différentes classes de figure. Les relations d'héritage sont basées sur la relation *est-un*, *i.e.* un `Rectangle` *est-une* `Figure`. En revanche, la relation de clientèle correspond à une relation a-un, *i.e.* une `Figure` (et donc toutes ses classes héritères) *a-un* `Point` d'origine.

1. En fait, l'accès est aussi autorisé à toutes les classes du package.



2 Relation d'appartenance

1) Écrivez une classe `Point` possédant deux attributs privées entiers représentant son abscisse et son ordonnée.

2) Ajoutez dans `Point` les constructeurs `Point` et les méthodes `abscisse` et `ordonnée` définis ainsi :

```
/**
 * Construit un point de coordonnées (a, b)
 *
 * @param a abscisse du point
 * @param b ordonnée du point
 */
public Point(double a, double b)

/**
 * Construit un point de coordonnées (0, 0)
 */
public Point()

/**
 * retourne l'abscisse du point
 */
public double abscisse()

/*
 * retourne l'ordonnée du point
 */
public double ordonnée()
```

3) Construisez une classe `Figure` contenant un attribut protégé `origine` de type `Point`.

4) Écrivez un constructeur pour la classe `Figure` qui initialise son origine à partir d'un point transmis en paramètre.

3 Relation d'héritage

5) Écrivez la classe `Rectangle` qui hérite de la classe `Figure`. Un `Rectangle` possède deux attributs entiers `longueur` et `largeur` en plus de son origine héritée de sa classe parente `Figure`.

```
public class Rectangle extends Figure {
  /**
   * longueur du rectangle
  */
  protected double longueur;

  /**
   * largeur du rectangle
   */
  protected double largeur;
}
```

6) Écrivez le constructeur de la classe `Rectangle` avec comme paramètres la largeur et la longueur, puis écrivez une classe de test et essayez de construire un rectangle. Quelle erreur de compilation obtenez vous ?

7) Au début du constructeur de `Rectangle` ajoutez l'appel au constructeur du parent :

```
super(new Point(0, 0));
```

de manière à initialiser un rectangle avec comme point d'origine $(0, 0)$.

8) En vous inspirant de votre cours, ajoutez les méthodes `périmètre`, `surface`, `changerLongueur` et `changerLargeur`.

9) Écrivez la méthode `toString` qui construit une chaîne de caractères qui représente un rectangle. Une unité de longueur est représentée par deux tirets horizontaux (`--`) et une unité de largeur par une barre verticale (`|`). Les quatre coins sont représentés avec des plus (`+`). Ainsi, un rectangle 10×3 sera représenté comme ci-dessous :

```
+--------------------+
|                    |
|                    |
|                    |
+--------------------+
```

10) Créez la classe `Carre` héritière de la classe `Rectangle`. Un carré ne possède pas d'autres attributs que ceux dont il hérite de ses ancêtres `Rectangle` et `Figure`.

11) Écrivez un constructeur qui initialise la longueur et la largeur d'un carré à partir de la longueur du côté transmise en paramètre. Ce constructeur devra faire appel à un constructeur de la classe `Rectangle`.

12) Dans votre classe de test, créez et affichez un carré. Changez la longueur du carré en faisant appel à la méthode `changerLongueur`. Affichez à nouveau le carré. Que remarquez-vous ?

13) Pour éviter ce problème, programmez dans la classe `Carre` deux nouvelles méthodes `changerLongueur` et `changerLargeur` qui mettent à jour, simultanément, la longueur et la largeur d'un carré. On dit que ces méthodes **redéfinissent** celles de la classe parente.

14) De manière similaire, écrivez une classe `Ellipse` définie par son grand axe et son petit axe qui hérite de `Figure`. Ajoutez une classe `Cercle` héritière de la classe `Ellipse`. On rappelle que pour une ellipse de petit axe a, et de grand axe b, le périmètre est égal à $\pi\sqrt{a^2b^2}$ et la surface πab.

4 Polymorphisme et liaison dynamique

Associés à celui d'héritage, les concepts de *polymorphisme* et de *liaison dynamique* donnent toute sa force à la programmation par objets. Dans ce TP, vous utiliserez les figures géométriques du TP précédent pour mettre en évidence ces deux mécanismes essentiels.

4.1 Polymorphisme

Le polymorphisme est la faculté pour une variable de désigner à tout moment des objets de types différents. En , le polymorphisme est contrôlé par l'héritage. C'est-à-dire qu'une variable peut désigner des objets de sa classe de déclaration, comme des objets de toutes les classes qui en dérivent. Par exemple, une variable de type `Rectangle` peut désigner un objet de type `Rectangle`, mais également de type `Carré`. Ceci est cohérent, dans la mesure où un carré est bien un rectangle (particulier). En revanche, elle ne pourra pas désigner des objets de type `Point` ou `Cercle`.

15) Placez dans la fonction `main` les deux déclarations suivantes, compilez votre classe de test, et vérifiez les affirmations précédentes.

```
Rectangle r1 = new Carre(8);
Rectangle r2 = new Cercle(8);
```

16) Remplacez la déclaration de la variable `r2` par :

```
Carre r2 = new Rectangle(2,8);
```

Compilez la classe de test. Quelle est la signification du message d'erreur ?

17) Remplacez la déclaration de la variable `r2` par :

```
Carre r2 = r1;
```

Compilez la classe de test. Pourquoi le compilateur indique-t-il une erreur alors que `r1` désigne une objet de type `Carre` ? Faites la conversion explicite de type demandée.

18) Déclarez un tableau de quatre figures et créez les quatre objets géométriques suivants :

```
Figure[] tf = new ...
tf[0] = new Figure(new Point(2,3));
tf[1] = new Rectangle(8,3);
tf[2] = new Carre(4);
tf[3] = new Cercle(4);
```

19) Appliquez une conversion de type explicite pour afficher la surface et le périmètre de chaque figure.

Les objets de type `Rectangle`, `Carre`, `Ellipse` ou `Cercle` peuvent tous retourner leur périmètre ou leur surface. En revanche, un objet de type `Figure` ne peut le faire puisque la classe `Figure` ne possède pas ces méthodes. L'instruction `tf[0].surface()` est donc invalide. D'ailleurs, si la classe `Figure` possèdait ces méthodes, elle ne saurait pas comment les programmer. En fait, la classe `Figure` ne permet pas de représenter des figures réelles, mais sert uniquement à définir les propriétés communes des figures géométriques. En , une telle classe est dite *abstraite*. Les méthodes qu'elle contient sont également dites abstraites. Ainsi, on pourra déclarer les signatures des méthodes `périmètre` et `surface`, mais sans en définir le corps. Une classe abstraite est donc ¡¡ incomplète ¿¿ puisque aucune action n'est associée à certaines méthodes. Il est impossible de créer, on dit aussi d'*instancier*, un objet d'une classe abstraite et la déclaration :

```
tf[0] = new Figure(new Point(0, 0));
```

devient invalide. En revanche, les classe dérivées devront définir les méthodes abstraites pour être ¡¡ complètes ¿¿ et pour instancier des objets.

20) Modifiez la déclaration de la classe `Figure` pour la rendre *abstraite* :

```
public abstract class Figure {
...
} //fin classe abstraite Figure
```

et ajoutez les *méthodes abstraites* :

```
/**
 * @return périmètre de la figure
 */
public abstract int périmètre();

/**
 * @return surface de la figure
 */
public abstract int surface();
```

21) Remplacez l'initialisation invalide de `tf[0]` par :

```
tf[0] = new Ellipse(3, 5);
```

Compilez votre classe de test et vérifiez que plus aucune erreur n'est signalée.

4.2 Liaison dynamique

22) Remplacez l'initialisation précédentes des quatre éléments du tableau de `Figures`, par une initialisation aléatoire. Vous utiliserez un générateur aléatoire de type `Random`, pour tirer au hasard une valeur comprise entre 0 et 3. Si cette valeur est zéro, vous créez une ellipse, si c'est un, vous créez un cercle, si c'est deux, vous créez un rectangle, et enfin si c'est trois, vous créez un carré.

23) À l'aide d'un énoncé itératif, affichez le périmètre et la surface des figures du tableau. Attention, la conversion de type explicite de type n'est plus possible, puisque vous ne connaissez plus de type exact de l'objet. Heureusement, cela est inutile. Dans l'écriture `tf[i].surface()`, la méthode `surface` est celle du type de l'objet crée et non celui du type de la variable qui le désigne, `Figure` en l'occurrence. Ce mécanisme s'appelle la *liaison dynamique*. D'une façon générale, lorsqu'il y a des redéfinitions de méthodes dans des classes parents, c'est à l'exécution que l'on connaît la méthode à appliquer. Elle est déterminée à partir de la forme dynamique de l'objet sur lequel elle s'applique.