Université Côte d'Azur
PeiP2

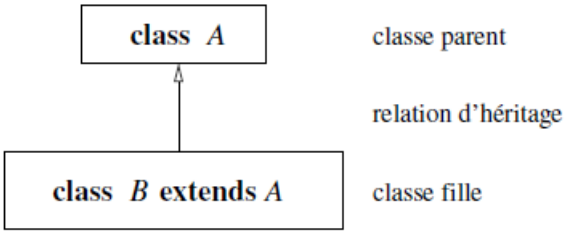
mercredi 6 octobre 2021
Durée: 3h

Algorithmique et Programamtion Java
Travaux Dirigés – Séance n. 5

---

1 Figures géométriques

En Java, 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 flêche 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 Java définit des 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].

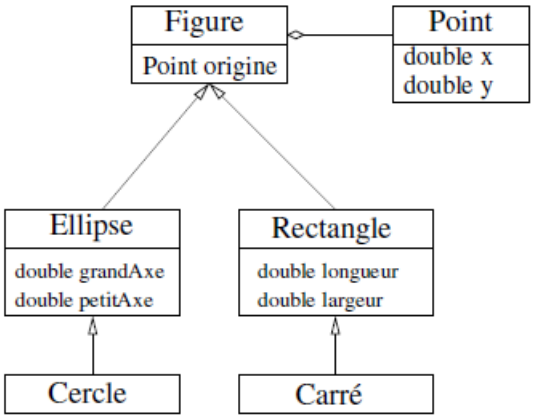
niveau	**accessibilité**
`private`	la classe de déclaration uniquement
`protected`	la classe de déclaration, ses héritières et le paquetage
`public`	toutes les classes

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

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

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

```
1  /**
2   * Construit un point de coordonnées (a, b)
3   *
4   * @param a abscisse du point
5   * @param b ordonnée du point
6   */
7  public Point(double a, double b)
8
9  /**
10  * Construit un point de coordonnées (0, 0)
11  */
12 public Point()
13
14 /**
15  * retourne l'abscisse du point
16  */
17 public double abscisse()
18
19 /*
20  * retourne l'ordonnée du point
21  */
22 public double ordonnée()
```

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

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

3 Relation d'héritage

exercice 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`.

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

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

exercice 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 ?

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

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

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

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

exercice 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 :

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

exercice 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`.

exercice 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`.

exercice 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 ?

exercice 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.

exercice 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, sa surface est égale à πab, et son périmètre est environ égal à $\pi\sqrt{2(a^2 + b^2)}$ (formule d'Euler).

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.

4.1 Polymorphisme

Le polymorphisme est la faculté pour une variable de désigner à tout moment des objets de types différents (bien que liés par des relations d'héritage). En Java, le polymorphisme est plus précisément contrôlé par l'héritage. C'est-à-dire qu'une variable peut tout aussi bien désigner des objets de sa classe de déclaration que 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`.

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

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

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

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

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

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

```
1 Carre r2 = r1;
```

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

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

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

exercice 19) Appliquez une conversion de type explicite pour afficher la surface et le périmètre de chaque figure, grâce à une boucle.

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 Java, 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 :

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

devient invalide. En revanche, les classe dérivées (si elles mêmse ne sont pas abstraites, mais concrètes) devront définir les méthodes abstraites pour être *complètes* et pour instancier des objets.

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

```
1 public abstract class Figure {
2 ...
3 } // fin classe abstraite Figure
```

et ajoutez les *méthodes abstraites* (prenez soin de placer le ; au bon endroit) :

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

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

exercice 21) Remplacez l'initialisation devenue invalide de `tf[0]` (pourquoi ?) 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

exercice 22) Remplacez l'initialisation précédente 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.

exercice 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 l'invocation de méthode s'applique.

5 Visualisation des formes

exercice 24) Ajoutez à vos classes la méthode `dessiner` qui dessine sur une planche à dessin, passée en paramètre, la forme géométrique courante.

exercice 25) Visualisez les formes géométriques de votre tableau.