INSA Centre Val de Loire
Département STI

Année 2018-2019

Programmation Orientée Objet

TD 3 – Héritage et Polymorphisme

Remarques :

1. Durant le TD, il est conseillé de faire la modélisation en utilisant le *plugin* Eclipse `eUML2`, et créer le corps des classes en Java ;
2. Pour utiliser ce plugin, on utilisera la version Indigo d'Eclipse disponible en `/ark/Logiciels/linux64/eclipse64/eclipse-indigo`
3. `eUML2` permet d'ajouter du code Java au moment de la modélisation et de créer un diagramme UML à partir de code Java.
4. On utilisera des affichages dans les diverses méthodes, constructeurs, afin de comprendre le fonctionnement interne des classes. Ces affichages ne sont mis dans ce contexte qu'à titre pédagogique et ne doivent pas être faits dans un cadre de production de code.
5. Ce TD s'étale sur deux séances.

À rendre à la fin de la seconde séance : sur le serveur Celene :

1. Une archive contenant le projet Eclipse
2. Un compte rendu en format pdf dans lequel vous répondez aux questions posées dans le sujet. Le symbole $\diamond$ signale qu'une question nécessite un commentaire dans le compte-rendu (il est possible que certaines question nécessitant commentaire n'aient pas de $\diamond$).

Exercice 1 : Une première série de classes

Télécharger le projet java Hiérarchie disponible sur moodle (cours Programmation Orientée Objet). Ce projet est composé du package `geometrie` et des classes `Point`, `Segment`, `Quadrilatere` et `TestTP` (cette dernière est utilisée pour tester votre programme Java) :

1. La classe `Point` est équipée des variables d'instance `x` et `y` de type `double` représentant les coordonnées cartésiennes (abscisse et ordonnée) d'un point dans un espace euclidien à deux dimensions, et d'une variable d'instance `label` de type `String` permettant d'identifier chaque point par une lettre (par exemple, `point A`). Cette classe a trois constructeurs :
   — un constructeur par défaut qui initialise le point à l'origine avec un label "O" (`(0, 0, O)`) ;
   — un constructeur qui initialise un point avec les coordonnées passées en paramètre ;
   — un constructeur prenant en paramètres les coordonnées cartésiennes du point et un *label*.

Chaque constructeur annonce l'existence du nouveau point (par affichage de celui-ci à l'écran). Pour mettre en pratique la réutilisation du code, on peut factoriser le code source en invoquant un constructeur à partir d'un autre (sans induire de boucle infinie).

2. La classe `Segment` est composée de deux `Point`s qui correspondent aux deux extrémités du segment de droite représenté. Le constructeur accepte deux `Point`s en paramètre. En plus des accesseurs pour chacune des extrémités, la classe définit les méthodes suivantes :
   — `longueur()` qui renvoie la longueur du segment ;
   — `toString()` qui surcharge la méthode de la classe `Object` et renvoie la chaîne de caractères `[AB]` où `A` et `B` sont les labels des deux extrémités.
3. La classe `Quadrilatere` est composée de quatre côtés (`Segment`). Elle possède deux constructeurs :
   — un constructeur par défaut qui initialise au quadrilatère trivial `OOOO` ;
   — un constructeur qui accepte quatre points en paramètre.
   La classe possède également la méthode `getPerimetre` qui renvoie la valeur du périmètre du quadrilatère (et qui affiche `Périmètre Quadrilatère`). Elle surcharge aussi la méthode `toString()` afin de pouvoir afficher le quadrilatère sous la forme : `[AB][BC][CD][DA]`.

Tester le projet :

a : ◇ À l'aide d'*e-UML2*, regarder l'architecture initiale du projet, ainsi que la composition des classes. (compte-rendu : le schéma UML obtenu)

b : ◇ Créer différentes instances de `Point` en utilisant les différents constructeurs. Commenter le fonctionnement de ceux-ci.

c : Créer un `Segment` à partir de deux points et afficher la longueur de celui-ci. Est-elle conforme à ce que vous pouvez attendre ? Corrigez.

d : Instancier un `Quadrilatere` à partir de 4 segments (et 4 points) et afficher son périmètre.

Exercice 2 : Héritage : classe `Rectangle`

a : Créer une classe `Rectangle` comme sous-type de `Quadrilatere`.

b : Définir un constructeur par défaut qui se contente d'afficher le message `Constructeur de Rectangle`

c : ◇ Instancier un rectangle. Tracer et expliquer les appels de constructeurs en analysant les messages affichés dans la console. Constater l'héritage de l'interface publique de `Quadrilatere` en affichant le périmètre du rectangle.

d : Créer un constructeur dans la classe `Rectangle` qui permet d'instancier un rectangle non trivial à partir de deux points (sommets opposés du rectangle – on fait l'hypothèse que les segments du rectangle sont parallèles aux axes du repère), en invoquant un constructeur de la classe `Quadrilatere`.

e : ◇ Donner le schéma UML obtenu.

Exercice 3 : Héritage : quelques classes supplémentaires

a : ◇ (Schéma UML) Enrichir la hiérarchie de classes en créant une classe `FigureGeometrique` dont héritent les classes `Polygone` et `Conique`. Ajouter les triangles et les pentagones en tant que sous-types de polygones. Rattacher les quadrilatères à la hiérarchie (sous-type de polygone). Créer les classes `Ellipse` et `Cercle` comme sous-types de `Conique`.

b : Ajouter deux variables membres `couleur` et `texture`, de type `String`, respectivement de visibilité `protected` et `private`, dans la classe `FigureGeometrique`. Écrire les accesseurs.

c : ◇ (Schéma UML) Ajouter un constructeur à la classe `FigureGeometrique` prenant comme paramètre une couleur et une texture. Modifier le code des classes filles en conséquence.

Exercice 4 : Polymorphisme

a : Créer une méthode générique `dessiner` dans la classe `FigureGeometrique` qui se contente d'afficher un message, déclinée en :

— `dessiner(int zone)`, qui affiche simplement *Dessin de la zone zone d'une figure géométrique.*

— `dessiner()`, qui affiche *Dessin d'une figure géométrique*

b : ◇ (Affichage lors de l'exécution puis commentaire) Ajouter une méthode `dessiner(Point p)` à la classe `Rectangle` qui affiche simplement le message *Dessin d'un rectangle.* Instancier un rectangle et vérifier le comportement de chacune des méthodes `dessiner` sur l'objet de type `Rectangle`.

```
Rectangle r = new Rectangle();
r.dessiner(4);
r.dessiner();
Point p = new Point();
r.dessiner(p);
```

c : Calculer le périmètre d'un objet de type `Rectangle`. Pour améliorer l'efficacité du calcul relatif au quadrilatère, redéfinir la méthode `getPerimetre()` dans la classe `Rectangle`. Cette dernière affichera en sus le message *Calcul du périmètre d'un rectangle.* Tester et constater l'exécution de la méthode redéfinie.

```
Quadrilatere quad = new Quadrilatere();
int perquad = quad.getPerimetre();

Rectangle rect = new Rectangle();
int perrect = rect.getPerimetre();
```

d : Proposer la redéfinition de la méthode `dessiner()` dans la classe `Rectangle`, affichant le message *Dessin d'un rectangle.*

e : ◇ (Affichage lors de l'exécution puis commentaire + questions) Tester le code suivant :

```
FigureGeometrique [] listFig = new FigureGeometrique[3];
listFig[0] = new Rectangle;
listFig[1] = new Quadrilatere();
listFig[2] = new Cercle();
```

```
for (int i=0;i<=2;i++) {
  listFig[i].dessiner();
}
listFig[0].getPerimetre();
```

Expliquer le principe de la liaison tardive pour l'évaluation de type.
Comment faire pour utiliser la méthode `getPerimetre()` de `Rectangle` sur listFig[0] ?
Est-il possible de déclencher la méthode `dessiner()` de `FigureGeometrique` pour les éléments de `listFig`, plutôt que les versions redéfinies dans les classes dérivées ?

f : ◇ Tester le code suivant :

```
Rectangle rec = new Quadrilatere();
```

Expliquer le message d'erreur à la compilation.

g : Dans la classe `FigureGeometrique`, déclarer une variable d'instance `code`, de type `int` et de visibilité `package`. L'initialiser à la valeur 0 dans les constructeurs. Dupliquer (c'est à dire re-déclarer) cette variable dans les classes `Quadrilatere`, `Rectangle` et `Cercle`, en l'initialisant, respectivement à 1, 2 et 3.

h : ◇ (Affichage lors de l'exécution puis commentaire + questions) Tester le code suivant :

```
for (int i=0;i <=2;i++)
   System.out.println(listFig[i].code);
```

Expliquer le principe de la liaison statique propre aux variables membres. À l'aide du transtypage, proposer un moyen d'atteindre pour l'objet de type `FigureGeometrique` la valeur des codes des sous-objets `Cercle`, `Quadrilatere` et `Rectangle`. Sans transtypage, comment récupérer la bonne valeur ?

Exercice 5 : Visibilité des méthodes et attributs

a : ◇ Ajouter une méthode **final** `dessiner()` à la classe `Quadrilatere`. Que se passe-t-il ?

b : ◇ Ajouter le marqueur **final** à la déclaration de la classe `Conique`. Constater et expliquer le message d'erreur à la compilation. Annuler la modification. À la lumière de ce constat, déterminer l'intérêt de l'usage du marqueur **final** pour les classes et les méthodes.

c : ◇ Restreindre la visibilité de la méthode `Quadrilatere.dessiner()` à **protected**. Constater et expliquer le message d'erreur à la compilation. Corriger.

d : ◇ Ajouter le marqueur `abstract` à la déclaration de la classe `FigureGeometrique`. Déclarer la méthode publique abstraite `toString()` dans `FigureGeometrique`. Expliquer le(s) message(s) d'erreur. Corriger les erreurs sans changer la classe `FigureGeometrique`.