UNIVERSITÉ CAEN NORMANDIE

U.F.R. Sciences

Université de Caen Normandie
Département mathématiques-informatique
2017–2018, première session
20 décembre 2017
Bruno Zanuttini

L2 informatique et L2 mathématiques

Contrôle terminal (correction)

Unité M.MIM3A1 : Introduction à la programmation orientée objet

2 h — Tous documents autorisés

---

1 Notions fondamentales (10 points)

Dans tous les exemples de code donnés, l'élision `[...]` représente des portions de code non pertinentes pour la question correspondante.

Question 1 (2 points) *Quelles sont les noms représentant des classes, et ceux représentant des instances, dans les lignes de code Java suivantes :*

```
[...]
X y = new X ();
z = y.f(t);
U v = null;
```

Correction Les noms de classes sont `X` et `U`, et les noms d'instances sont `y`, `z`, `t` et `v`.

Question 2 (2 points) *On considère les définitions de classes Java suivantes :*

```
public class A {
    public void f () {System.out.println("Je suis A");}
    public void g () {f(); f();}
}

public class B extends A {
    @Override
    public void f () {System.out.println("Je suis B");}
}
```

Que vont afficher les instructions suivantes ? Justifier brièvement.

1. `A a = new A (); a.g();`
2. `B b = new B (); b.g();`

3. `A c = new B (); c.g();`
4. `A a = new A (); B b = new B (); a = b; a.g();`

Correction

1. `Je suis A`
   `Je suis A`
   En effet, `a` est une instance de `A`, donc c'est la méthode `f` de `A` qui est appelée
2. `Je suis B`
   `Je suis B`
   En effet, `b` est une instance de `B`, donc c'est la méthode `f` de `B` qui est appelée
3. `Je suis B`
   `Je suis B`
   En effet, `c` est une instance de `B`, donc même si elle est déclarée de type `A`, c'est la méthode `f` de `B` qui est appelée
4. `Je suis B`
   `Je suis B`
   En effet, la variable `a`, même si elle est déclarée de type `A`, contient une instance de `B`, donc c'est la méthode `f` de `B` qui est appelée

Question 3 (2 points) *Dans le code Java suivant, le développeur a cherché à éviter la redondance de code entre ses classes* `Point` *et* `PointAvecTexte`*, en ajoutant la relation d'héritage. Quelle erreur de conception a-t-il commise ? Justifier brièvement.*

```
public class Point {
    protected int x;
    protected int y;
    [...]
    @Override
    public String toString () {
        return "(" + this.x + "," + this.y + ")";
    }
}

public class PointAvecTexte extends Point {
    protected int x;
    protected int y;
    protected String texte;
    [...]
    @Override
    public String toString () {
        return "(" + this.x + "," + this.y + ":" + this.texte + ")";
    }
}
```

Correction Les attributs `x` et `y` sont redeclarés dans la sous-classe `PointAvecTexte`, alors qu'ils sont hérités de la classe mère `Point`.

Question 4 (2 points) *On suppose que l'on a une interface* `I`*, une classe abstraite* `A` *et trois classes (non abstraites)* C_1*,* C_2 *et* C_3*, telles que toutes les classes possèdent un constructeur sans argument, et*

- `A` *implémente* `I`*,*
- C_1 *et* C_2 *héritent de* `A`*,*
- C_3 *implémente* `I`*,*

à l'exclusion de toute autre relation entre ces classes. Dire, pour chacune des instructions suivantes, si elle a du sens ou non (dans ce dernier cas, justifier brièvement) :

1. `I i = new I ();`
2. `A a = new A ();`
3. C_1 `c1 = new` C_1 `();`
4. C_3 `c3 = new` C_3 `();`
5. `I i = new A ();`
6. `I i = new` C_1 `();`
7. `I i = new` C_3 `();`
8. `A a = new` C_1 `();`
9. `A a = new` C_3 `();`
10. C_1 `c1 = new` C_2 `();`
11. C_1 `c1 = new` C_3 `();`

Correction

1. Non, une interface ne peut pas être instanciée
2. Non, une classe abstraite ne peut pas être instanciée
3. Oui
4. Oui
5. Non, une classe abstraite ne peut pas être instanciée
6. Oui
7. Oui
8. Oui
9. Non, `A` n'est pas une classe ancêtre de C_3
10. Non, C_1 n'est pas une classe ancêtre de C_2
11. Non, C_1 n'est pas une classe ancêtre de C_3

Question 5 (2 points) *On considère la classe Java suivante :*

```
public class C {
    private static int attribut = 0;
    public void f ([...]) {
        C.attribut = C.attribut + 1;
        [...]
    }
    public static int g () {
        return C.attribut;
    }
}
```

Comment interpréter la valeur retournée par la méthode `g` *lorsqu'elle est appelée ?*

Correction La méthode `g` retourne le nombre de fois où `f` a été appelée sur une instance, quelle qu'elle soit, de la classe `C`.

2 Conception (10 points)

On souhaite réaliser un *package* Java permettant de représenter des formes géométriques régulières en deux et trois dimensions, et de calculer leur surface et leur volume, respectivement. On rappelle que la surface d'un disque de rayon r est πr^2, que celle d'un rectangle de côtés c_1, c_2 est $c_1 \times c_2$, que le volume d'un cylindre de hauteur h et ayant pour base un disque de rayon r est $\pi r^2 \times h$, que celle d'un parallélépipède de base $c_1 \times c_2$ et de hauteur h est $c_1 \times c_2 \times h$, et que celle d'une boule de rayon r est $\frac{4}{3}\pi r^3$.

Pour toutes les questions qui suivent, on pourra répondre avec du code Java ou avec du pseudo-code, au choix. Pour les questions 6 et 7, seul le squelette des classes et interfaces est demandé (attributs, et méthodes avec type des arguments et de retour), pas le corps des méthodes.

Question 6 (2 points) *Sachant qu'on ne s'intéresse qu'au calcul des surfaces et volumes, proposer une interface* `Figure2D` *permettant de représenter une figure régulière quelconque en deux dimensions, ainsi qu'une interface* `Figure3D`*. Pour cela, lister les méthodes des interfaces, avec pour chacune ses arguments et son type de retour. Proposer une classe* `Disque` *et une classe* `Rectangle` *implémentant l'une et/ou l'autre de ces interfaces, en listant de même leurs attributs et méthodes.*

Correction Dans `Figure2D`, une méthode `surface()` permettant de calculer la surface, ne prenant pas d'argument et renvoyant une valeur (par exemple un `float` en Java), et une méthode similaire `volume()` dans `Figure3D`. Les classes `Disque` et `Rectangle` implémenteraient l'interface `Figure2D`, n'auraient pas de lien autre entre elles, et pas de lien avec l'interface `Figure3D`. La classe `Disque` comporterait un attribut `rayon`, un constructeur prenant ce rayon en argument, et définirait la méthode `surface()` déclarée par l'interface `Figure2D`. De même, la classe `Rectangle` comporteait des attributs `côté1` et `côté2`, un constructeur prenant deux valeurs pour ces attributs en arguments, et définirait la méthode `surface()`.

Question 7 (4 points) *Compléter la conception précédente avec des classes* `Pallélépipède`*,* `Cylindre` *et* `Boule`*. On veillera à éviter au maximum la duplication de code (même logique répétée plusieurs fois), en introduisant si besoin des classes et interfaces additionnelles. Pour chaque classe ou interface proposée, donner les attributs, les méthodes déclarées, définies et redéfinies en expliquant leurs arguments, leurs valeurs de retour et leur fonctionnement, et donner les liens entre les différentes classes et interfaces (implémentation, héritage, utilisation d'instances de l'une comme attributs d'une autre, etc.).*

Correction On peut proposer par exemple des interfaces `Figure2D` et `Figure3D`, respectivement, comme à la question précédente. On peut ensuite représenter les figures en deux dimensions par autant de classes concrètes implémentant toutes l'interface `Figure2D`. Pour les figures en trois dimensions, on peut constater que le volume d'un cylindre et celui d'un parallélépipède sous tous les deux calculés en multipliant la surface de la base par la hauteur. On peut donc factoriser ce calcul dans une classe (abstraite ou non) `Figure3DDroite` implémentant l'interface `Figure3D`, dont hériteraient deux classes concrètes `Parallélépipède` et `Cylindre`. En revanche, la classe `Boule` implémenterait directement l'interface `Figure3D`.

On peut proposer notamment une classe `Figure3DDroite` concrète comportant un attribut de type `Figure2D` et un attribut numérique `hauteur`, les sous-classes fixant simplement le type de figure à deux dimensions, ou encore une classe `Figure3DDroite` abstraite, possédant une méthode abstraite `surfaceBase()`, à définir par les sous-classes.

Dans le premier cas, le constructeur de `Figure3DDroite` prendrait en argument un objet de type `Figure2D` pour sa base et une valeur numérique pour la hauteur, et sa méthode `volume()` pourrait être implémentée en multipliant le résultat de l'appel `surface()` sur sa base, par sa hauteur. La sous-classe `Parallélépipède` n'aurait besoin que d'un constructeur, prenant par exemple deux valeurs `côté1` et `côté2` en arguments et appelant simplement `super(new Rectangle(côté1, côté2))`, et de même pour la classe `Cylindre`.

Dans le second cas, le constructeur de `Figure3DDroite` prendrait simplement une valeur pour la hauteur, et sa méthode `volume()` serait implémentée en multipliant le résultat de l'appel `this.surfaceBase()` par sa hauteur. La méthode abstraite `this.surfaceBase()` serait définie par les sous-classes `Parallélépipède` et `Cylindre`, qui comporteraient par exemple un attribut de type `Rectangle` ou `Disque`, respectivement.

Dans les deux cas, la classe `Boule` implémenterait directement l'interface `Figure3D`, en prenant à la construction une valeur pour son rayon, et en définissant la méthode `surface()`.

Question 8 (2 points) *Parmi les types proposés, donner le type à utiliser pour une liste contenant*

1. *des figures en trois dimensions quelconques,*
2. *des parallélépipèdes et des cylindres,*
3. *des cylindres et des boules,*
4. *des cylindres et des disques.*

Dans chacun des ces quatre cas, répondre en donnant le type `T` *le plus spécifique possible, parmi ceux proposés précédemment, tel que le type Java* `List<T>` *permette de contenir les objets cités.*

Correction

1. `Figure3D`
2. `Figure3DDroite`
3. `Figure3D`
4. `Object`

Question 9 (2 points) *Que prendrait en argument une méthode permettant de calculer le volume total occupé par une liste de figures quelconques en trois dimensions (parmi celles considérées) ? Quel algorithme utiliseriez-vous pour définir cette méthode ?*

Correction La méthode prendrait une liste d'objets de type `Figure3D`. Elle initialiserait son résultat à 0, puis parcourrait cette liste en incrémentant à chaque fois le résultat de la valeur retournée par l'appel de la méthode `volume()` sur la figure courante.