

INTRODUCTION A LA PROGRAMMATION

Corrigé de l'examen final

Exercice 1 : Conception OO et programmation [50 points]

Correction : voir le fichier `Simulation.java`.

See the file `Simulation.java`.

BARÈME : Le détail du barème pour la conception est donné en commentaire dans ce fichier. The details of the grading schema are provided as comments in the file `Simulation.java`.

Exercice 2 : Concepts de base[42 points]

1. [6 points]

- (a) Le code affiche

2 4

Justification : la méthode `m` a pour paramètres un objet `a` de type `A` et un entier `i` (type de base). Les deux paramètres sont passés par valeur car seul le passage par valeur existe en java. Il est possible dans `m` de changer le contenu référencé par le paramètre `a` mais pas la référence `a` elle-même. Les modifications faites dans `m` de `i` et de `a` ne se répercutent pas à l'extérieur par contre la modification de l'objet référencé est visible.

- (b) Le code affiche

2 3

Justification : en supprimant `this.` le setter affecte une valeur à son paramètre et non à l'attribut `a`.

Détail barème : donné directement dans la fiche de correction.

2. [9 points]

```
// in class A
@Override
    public String toString(){return "" + a;}
// in class B
@Override
    public String toString(){ return super.toString() + " " + b;}
```

Détail barème : donné directement dans la fiche de correction.

3. [15 points]

- (a) Le programme affiche :

10
20
10
20
2

Justification : la ligne 25 appelle le constructeur de la ligne 4 qui appelle celui de la ligne 8. Ce dernier affiche la valeur de l'attribut `w` auquel a été affectée la valeur 10. Le corps du constructeur de la ligne 4 affiche la valeur de `w` multiplié par 2 ; c'est-à-dire 20. La ligne 26 appelle le constructeur de la ligne 18 qui appelle celui de la ligne 4. Ce dernier va afficher les même valeur que précédemment. le corps du constructeur de la ligne 18 affiche la valeur de l'attribut `x` qui vaut 2.

- (b) Les lignes 18 et 25 émettront un message d'erreur car elles font toutes deux appel au constructeur par défaut de `W` qui a disparu du moment qu'il y a un constructeur explicite en ligne 8.

- (c) Le programme affiche :

10
10
2

Justification : la ligne 25 appelle le constructeur de la ligne 4 uniquement qui la valeur de `w*2` qui est initialisé à 5 par défaut (affichage de 10). La ligne 26 appelle ce même constructeur implicitement puis affiche la valeur de l'attribut `x` comme auparavant.

- (d) l'attribut `w` de la super-classe est privé. Il ne peut pas être accédé dans la super-classe. Pour initialiser `w` à 12 il faut que le constructeur de la sous-classe s'écrive

```
public X(){super(12); System.out.println(x);}


```

- (e) oui, la sous-classe `X` aurait alors deux attributs `w` (le sien spécifiquement et celui hérité de plus haut). L'affichage reste inchangé. Le `w` affiché dans la classe `W` reste le même et ceux faits dans `X` restent inchangés.

4. [12 points]

- (a) La ligne 36 appelle la méthode `sing` sur un objet dont le type apparent est `Artist`. S'il n'y a pas de méthode `sing` dans la class `Artist` le compilateur ne trouvera pas quelle méthode appliquer.

- (b) L'instanciation de la ligne 53 ne pourra plus être faite car la classe `Painter` devient abstraite.

- (c) la méthode `perform` ne tire pas parti du polymorphisme et fait des tests de type. Elle devra être modifiée à chaque fois que l'on ajoute un type d'acteur et/ou une activité spécifique à un type d'acteurs. Par ailleurs, la méthode `sing` ne fait peut-être pas sens pour un `Painter` (ou la méthode `paint` pour un `Singer`). Il n'y a pas de raison d'imposer ces méthodes à tout type d'artiste. Pour y parer, il suffirait de définir une méthode abstraite, `perform`, dans `Artist` et la redéfinir spécifiquement dans les sous-classes. Elle appellerait `sing` dans `Singer` et `paint` dans `Painter` par exemple. La méthode `RiveGauche::perform` consisterait alors à boucler sur tous les éléments de la collection d'acteurs en appelant leur méthode `perform` (ce qui se fera de façon polymorphe vu que l'on a une collection de pointeurs).

- (d) Il est possible par exemple d'introduire une interface

```
interface Exposable {  
    List<String> listExpositionPlace();  
}
```

et qui serait implémentée par les classes modélisant les peintres et les sculpteur.

Exercice 3 : Déroulement de programme [13 points]

Le programme affiche :

```
smallGREEN:smallGREEN->false  
smallGREEN:bigRED(0)->false  
smallGREEN:bigRED(1)->false  
bigRED(0):smallGREEN->true  
bigRED(0):bigRED(0)->false  
bigRED(0):bigRED(1)->false  
bigRED(1):smallGREEN->true  
bigRED(1):bigRED(0)->true  
bigRED(1):bigRED(1)->false
```

Justifications :

1. (1 point) Le programme principal crée une collection hétérogène d'objets de types A et y met un B et deux C (sous-classes de DA).
2. (2 points) Les lignes 79 à 83 font interagir pair à pair chaque élément de la collection au moyen de leur méthode e, et indique qui interagit en affichant un identifiant créé au moyen de la méthode d.
3. (2 points) la méthode d d'un B affiche smallGREEN celle d'un C bigRED(0) (pour le premier C et bigRED(1) pour le second (entre parenthèse la valeur de leur attribut h).
4. (3 points) l'interaction simule le «double dispatch» au moyen de la surcharge. Si on remplace e par "peut manger" (voir étude de cas sur la polymorphisme et la simulation du «double dispatch»), l'exécution reflète le fait : qu'un B ne peut manger ni un B ni un C, que tout C peut manger un B et qu'un C peut manger un autre C que s'il est plus "fort" que lui (h plus grand).

Le code simule donc un mécanisme de «double dispatch» au moyen de la surcharge de méthode (comme dans l'exemple «prédateur-proie» vu en étude de cas du cours).

BARÈME :

- 1 point par ligne d'affichage correcte
- Pour la justification 8 points (voir les points par éléments d'explication suggéré dans le corrigé ci-dessus). Si les explications sont plus verbeuses et/ou très différentes dans le forme du corrigé, donnez les points au pro rata des lignes de code (bien) expliquées.