Université de Tlemcen
Faculté des Sciences
Département d'Informatique
2e année de licence

Corrigé-type de l'Examen final de Programmation Objet

Année universitaire 2016-2017
Jeudi 26 janvier 2017
Durée : 1h30

Dans les questions à choix multiples (QCM), cochez une seule réponse.

Questions de cours (15 min ; 2,5 pts) (réponse correcte +0,5 / incorrecte -0,1)

1) Héritage

a) Quand on conçoit une hiérarchie de classes, que peut-on dire d'une superclasse ?

✔ Une superclasse doit contenir les données et fonctionnalités qui sont communes à toutes les sous-classes qui héritent de la superclasse.
Une superclasse doit être la classe la plus grande, la plus complexe, dont toutes les autres sous-classes sont dérivées.
Une superclasse doit contenir les données et fonctionnalités qui sont uniquement requises pour la classe la plus complexe.
Une superclasse doit avoir des données publiques pour fournir l'accès à toute la hiérarchie de classes.
Une superclasse doit contenir les détails les plus spécifiques de la hiérarchie de classes.

b) Parmi les affirmations suivantes concernant la conception de classes, cochez celle qui est **fausse** :

Si une classe C1 a une variable d'instance dont le type est une autre classe C2, alors un C1 ***a un*** C2.
✔ Si deux classes C1 et C2 ont une relation telle que un C1 ***est un*** C2, alors un C2 ***a un*** C1.
Si la classe C1 est indépendante, alors aucune de ses méthodes n'a de paramètres qui sont des instances d'une autre classe.
Des classes qui ont des méthodes en commun ne définissent pas forcément une relation d'héritage.
Si la classe C2 hérite de la classe C1, la classe C1 ne dépend pas de C2 pour son implémentation.

2) Interfaces

a) Une interface `Affichable` écrite en Java contient une seule méthode, `afficher()`. Une classe `Figure` implémente cette interface. Quelle affirmation parmi les suivantes est vraie ?

`Figure` doit obligatoirement redéfinir `afficher()`.
✔ Pour que `Figure` soit instanciable, il faut qu'elle contienne une redéfinition de `afficher()`.
Il est possible que `Figure` ne soit pas abstraite et ne redéfinisse pas `afficher()` : il suffit que la méthode `afficher()` dans `Affichable` ne soit pas abstraite.
Si `Figure` est une classe abstraite, il ne faut pas qu'elle redéfinisse `afficher()`.
Pour que `Figure` soit instanciable, elle ne peut implémenter qu'une seule interface (pas d'héritage multiple).

b) On modifie l'interface `Affichable` en y ajoutant une autre méthode, `masquer()`, mais on ne modifie pas la classe `Figure`. Quelle conséquence cela aura-t-il sur `Figure` ? [*cochez seulement la réponse la plus correcte*]

Aucune conséquence.
✔ Aucune conséquence si `Figure` est une classe abstraite.
Il est possible que cela n'ait aucune conséquence : il faut que `masquer()` dans `Affichable` ne soit pas abstraite.
`Figure` ne peut plus être compilée. Sa compilation provoque une erreur.
Pas d'erreur de compilation, mais `Figure` ne peut plus être instanciée, alors que c'était possible avant.

3) [*Cochez la réponse **fausse**.*] Une classe d'objets immuables...

...peut définir des getters dans certains cas.
...peut définir des getters s'ils ne concernent que les champs objets immuables ou de type primitif.
...ne doit pas définir des setters, même s'ils ne concernent que les champs objets immuables.
✔ ...doit être déclarée avec le mot-clé **`immutable`**.
...définit un type d'objets dont l'état ne peut pas être modifié.

Exercice 1 (20 min ; 3,5 pts)

Soit la classe `Parking` qui est définie avec la variable d'instance suivante :

```
private List<Voiture> voitures = new ArrayList<>();
```

La classe `Voiture` a plusieurs champs. Une de ses méthodes, `getMarque()`, renvoie la marque.

1) Écrire le corps de la méthode `sortir()` définie dans `Parking`, qui permet de supprimer de la liste toutes les voitures d'une marque donnée et qui renvoie la liste des voitures supprimées : **(3 pts)**

```
// Attention (rappel) : les indices des éléments d'une liste peuvent changer !
public List<Voiture> sortir(String marque){
    List<Voiture> resultat = new ArrayList<Voiture>();
    for (int i=0 ; i < voitures.size() ; ++i){ // for-each -> faux
        Voiture v = voitures.get(i);           // while avec itérateur -> OK mais
        if (v.getMarque().equals(marque)){     // alors utiliser la méthode remove()
            resultat.add(v);                   // de l'itérateur...
            voitures.remove(i);                // <- pas celle-ci
            i--;   // l'indice du prochain élément a changé
        }
    }
    return resultat;
}
```

2) La classe `Voiture` peut-elle être abstraite ? Expliquez (en quelques mots) **(0,5 pt)** :

Oui, elle peut être abstraite. Il faut juste qu'il y ait des sous-classes de Voiture, dont les instances pourront être stockées dans la liste (grâce au principe de la substitution).

Exercice 2 (55 min ; 14 pts)

1) Quel principe fondamental de l'approche orientée objet n'a pas été respecté dans la classe `Compteur` ci-contre [*répondre par un seul mot*] ? **(0,5 pt)**

```
public class Compteur{
    public int valeur;
}
public class Machine{
    private Compteur actions = new Compteur();

    public void actionner(){
        actions.valeur++;
    }
}
```

encapsulation

(**masquage des données** → 0,25 pt)

2) Pour respecter ce principe, dans la déclaration du champ `valeur`, on remplace le mot-clé `public` par `private` et on recompile les deux classes. Quelle conséquence aura ce changement sur la méthode `actionner()` de `Machine` ? **(+0,5 pt / -0,1 pt)**

- ✔ Erreur à la compilation.
- Pas d'erreur à la compilation mais une erreur à l'exécution.
- Pas d'erreur à la compilation ni à l'exécution mais erreur logique.
- Autre erreur.

3) On veut éliminer l'erreur. Pour cela, **a)** il est d'abord utile d'ajouter dans la classe `Compteur` : **(+0,5 pt / -0,1 pt)**

- ✔ un accesseur `getValeur()` et un mutateur `setValeur()`.
- un accesseur seulement (un mutateur n'est pas nécessaire).
- un mutateur seulement (un accesseur n'est pas nécessaire).
- un constructeur.
- rien (il est inutile de modifier `Compteur`).

b) Ensuite, l'instruction de `actionner()` dans `Machine` est remplacée par l'instruction suivante (*écrivez une seule instruction*) : **(0,5 pt)**

```
actions.setValeur(actions.getValeur()+1);
```

4) On ajoute la méthode suivante à `Machine` :

```
public void afficheCompteur(){
    System.out.print(actions);  // (print, pas println)
}
```

Qu'affiche-t-elle à l'écran ? **`Compteur@XXXXX`** ***(avec XXXXX = référence de l'objet*** **`actions`***)*** .....**(0,5 pt)**

5) Que faut-il faire dans la classe `Compteur` pour que `afficheCompteur()` affiche bien la valeur du compteur ?

a) ***Il faut*** **redéfinir la méthode `toString()`**.....**(0,5 pt)**

b) Faites-le : .....**(1 pt)**

```
public String toString(){
    return  "" + valeur; // ou String.valueOf(valeur);
}
```

6) Que donne l'exécution des instructions ci-contre ? **(2 pts)**

`101Exception …`
L'appel de machines[3].afficheCompteur() provoque une exception de type NullPointerException

(« 101 » → 1,5 pt / Exception → 0,5 pt)

```
Machine[] machines = new Machine[4];
machines[0] = new Machine();
machines[1] = new Machine();
machines[2] = machines[0];
machines[2].actionner();
for (Machine m : machines)
    m.afficheCompteur();
```

7) On commence à définir la classe `Turbine` de la manière suivante :

```
public class Turbine extends Machine{
    private Compteur vitesse;
    public Turbine(){
        vitesse = new Compteur();
    }
```

a) Écrire le corps de sa méthode `actionner()` pour qu'elle augmente de **2** la valeur du compteur `actions` et affecte la valeur donnée en paramètre au compteur `vitesse` : **(1,5 pts)**

```
    public void actionner(int vitesse){
        actionner();  // ou super.actionner()
        actionner();
        this.vitesse.setValeur(vitesse);
    }
```

b) Écrire le corps de sa méthode `afficheCompteur()` pour afficher les valeurs des compteurs comme ceci :

```
Actions : 4 – Vitesse : 150
```

(1,5 pts)

```
    public void afficheCompteur(){
        System.out.print("Actions : ");
        super.afficheCompteur();
        System.out.println(" - Vitesse : " + vitesse);
                                    // ou vitesse.toString() ou vitesse.getValeur();
    }
```

c) Soit les déclarations suivantes :

```
Turbine t1 = new Turbine(), t2 = new Turbine(), t3 = new Turbine();
Machine m1 = new Machine(), m2 = new Machine(), m3 = new Machine();
```

Les instructions suivantes sont elles correctes, ou donnent-elles une erreur ? **(2 pts / -0,5 pts)**

`t1 = m1;`	☐ correcte	☑ erreur à la compilation	☐ erreur à l'exécution **(+0,4 / -0,1)**
`t2 = (Turbine) m2;`	☐ correcte	☐ erreur à la compilation	☑ erreur à l'exécution **(+0,4 / -0,1)**
`m1 = t1;`	☑ correcte	☐ erreur à la compilation	☐ erreur à l'exécution **(+0,4 / -0,1)**
`m1 = t3; t1 = m1;`	☐ correcte	☑ erreur à la compilation	☐ erreur à l'exécution **(+0,4 / -0,1)**
`m2 = t3; t1 = (Turbine) m3;`	☐ correcte	☐ erreur à la compilation	☑ erreur à l'exécution **(+0,4 / -0,1)**

d) Soit la déclaration suivante :

```
Machine v = new Turbine(), w = new Turbine();
```

Que donne l'exécution des instructions suivantes ?	***Résultat affiché ou type d'erreur***	**(1 pt)**
`w.actionner(); w.afficheCompteur() ;`	**`Actions : 1 – Vitesse : 0`**	**(0,5 pt)**
`v.actionner(99); v.afficheCompteur();`	**Erreur à la compilation (actionner(int) absent dans Machine)**	**(0,5 pt)**

e) Peut-on dire que la méthode `actionner()` et la méthode `afficheCompteur()` sont **redéfinies** dans la classe `Turbine` ? Expliquez **(0,5 pts)** :

Non. Seule la méthode afficheCompteur() est redéfinie (même signature). La méthode actionner() est surchargée (même nom mais signatures différentes).

8) On veut ajouter un constructeur à `Machine` qui initialise son compteur en utilisant une valeur entière donnée en paramètre.

a) Écrivez sa définition **(0,5 pts)** :

```
public Machine(int actions){
    this.actions.setValeur(actions);
}
```

b) Après cette modification, on veut recompiler toutes les classes du projet et les tester. Un problème apparaît. Dites lequel, et expliquez comment le résoudre exactement **(1 pt)** ?

Erreur à la compilation de `Turbine`. (Comme son constructeur n'a pas d'appel explicite au constructeur de la superclasse, le compilateur a ajouté `super();` dans sa 1re ligne. Quand on a ajouté un constructeur avec paramètre à `Machine`, celle-ci n'a plus le constructeur par défaut. L'appel `super()` provoque donc une erreur.)

Solution : ajouter `super(0);` au début du constructeur de `Turbine` ou bien ajouter un constructeur sans paramètre dans `Machine`.

Bon courage.

Amine Brikci-Nigassa
nh2@libretlemcen.org
nh2blog.wordpress.com
twitter.com/nh2