

# INTRODUCTION A LA PROGRAMMATION

## Corrigé test semestre 1

### Exercice 1 : Programmation [15 points]

Toute solution valable est acceptée (y compris avec des éléments de l'API ou des nouveautés de Java nous vues en cours), comme celle-ci :

```
public static int[][] enlarge(int[][] image, int[] seam) {
    int width = image[0].length;
    int height = image.length;
    int[][] result = new int[height][width + 1];
    for (int y = 0; y < height; ++y) {
        System.arraycopy(image[y], 0, result[y], 0, seam[y] + 1);
        System.arraycopy(image[y], seam[y], result[y], seam[y] + 1, width - seam[y]);
    }
    return result;
}
```

### Exercice 2 : Conception OO et programmation [42 points + 8 bonus]

Il existe bien sûr de nombreuses variantes acceptables pour les parties 1 et 3. (pour une variante possible, voir le fichier source `RobotSimulation.java`)

Pour la description de la classe en charge de la simulation :

1. il faudrait une classe `RoomSimulation` implémentant l'interface `Simulation`
2. cette dernière aurait comme attribut une `Room` et serait en charge d'invoquer `move` pour tous les aspirateurs de cet attribut.
3. suite à l'invocation de `move` la simulation doit éventuellement mettre à jour le nombre de dalles propres (`addClean` dans la conception proposée plus haut) et supprimer les objets aspirés de la pièce (`removeItem`).
4. la fin de la simulation est décidé au moyen de `Room.isSolved`

Pour la partie 3 :

1. la méthode de `DaedalusSimulation` qui invoque la méthode `move` sur tous les «animaux» devrait être capable de distinguer les prédateurs des proies (pour savoir quoi faire si deux animaux se rencontrent suite à un `move`);
2. une façon de faire consiste à mimer le «double dispatch» au moyen de la surcharge de méthodes (tel que vu dans l'étude de cas en cours: scorpion mangeant une souris). Des méthodes telles que `canKill` et `isKilledBy` devraient être ajoutées dans la hiérarchie d'animaux.

### Exercice 3 : Concepts [35 points]

Répondez clairement et succinctement aux questions suivantes :

1. [6 points]
  - (a) 

```
public A(){
    this(1,1);
}
```

- (b) il est utile car les deux instructions suivantes n'initialisent pas les mêmes `a1` et `a2`.
- (c) 

```
public void display() {  
    System.out.println(a1 + " " + a2 + " "+ super.a1 + " " + super.a2);  
}
```
2. [4 points]
- (a) oui, la surcharge étant la définition de deux méthodes de même nom avec des listes de paramètres différents)
- (b) par exemple `public static int small(int number)` (le type de retour ne fait pas partie du mécanisme de surcharge).
3. [ 6 points]
- (a) Le constructeur de `A` lance une «checked exception». Ce constructeur est appelé par celui de `B` qui ne traite ni ne déclare l'exception.
- (b) en modifiant l'entête du constructeur de `B` comme suit :
- ```
public B(int a, int b) throws E
```
- A noter, qu'il n'est pas possible de mettre l'appel à `super` dans un `try` car `super` doit être la première instruction.
- (c) `E` devrait au minimum contenir les deux constructeurs (un par défaut et un prenant en paramètre une `String`) pour préserver le fonctionnement usuel de `getMessage`
4. [ 5 points]
- (a) La syntaxe de l'appel `I.m()` ; ne s'applique qu'aux éventuelles méthodes statiques de l'interface. Elle n'est pas utilisable pour les méthodes avec définition par défaut.
- (b) la ligne 8 devrait s'écrire `I.super.m()` ;
- (c) non car les méthodes d'interface sont nécessairement `public`
- (d) non car on ne peut pas restreindre les droits d'une méthode implémentant une interface.
5. [ 5 points] A l'exécution de `A a = new B(2,3)`, le programme affiche :

L'objet `B` vaut : 2 0

Au moment où le constructeur de `B` invoque celui de `A` l'attribut `b` n'est pas encore initialisé explicitement par le constructeur de `B` il a donc sa valeur par défaut (zéro). Il n'est pas recommandé d'invoquer des méthodes polymorphiques dans les constructeurs (ici `toString`) car elles peuvent alors agir sur des objets partiellement initialisés).

6. [ 6 points] Le programme affiche :

```
est-ce ainsi  
que les hommes vivent ?
```

Justification principale: Java utilise le passage par valeur. La fonction `f` manipule une copie de son premier argument `s1`. Les modifications faites sur `s1` dans `f` ne se répercutent pas à l'extérieur de la fonction. La chaîne passée en argument reste inchangée. La fonction `g` utilise aussi le passage par valeur. Le premier argument qui est une référence reste inchangé mais l'objet qu'il référence peut par contre être modifié. Les différentes méthodes invoquées dans `g` remplacent le contenu initial de la première case de `s3` en la chaîne `que les hommes vivent ?`.

7. [ 3 points] Les méthodes avec définition par défaut permettent essentiellement de pouvoir modifier le contenu d'interfaces existantes sans que cela n'ait d'impact sur la compilabilité des classes qui les implémentent.

### Exercice 3 : Analyse de programme [18 points]

1. Java met systématiquement en place la résolution dynamique des liens. Une moto stockée dans une variable de type voiture sera vue comme une moto lors du calcul du prix. La contrainte qui veut qu'une moto se parquant sur une place de voiture paie le tarif d'une voiture n'est pas respectée.
2. Les dimensions des tableaux sont reprises «en dur» (risque d'incohérence lors de modifications), le code ne vérifie pas les cas limites (objet `null`, place déjà prise )
3. L'un des principaux défauts de cette conception est de confondre les places et les véhicules. Certains éléments du calcul du prix sont en effet liés à la place et d'autres, comme la réduction, au véhicule. Il faut donc deux hiérarchies distinctes une pour les véhicules et une autre pour les places. Le fichier `ParkingCorrige.java` donne une conception alternative possible corrigeant les problèmes précités.