

INTRODUCTION A LA PROGRAMMATION

Test Semestre I

Instructions :

- Vous disposez de une heure quarante cinq minutes pour faire cet examen (8h15 - 10h);
- Nombre maximum de points 110 points (dont environ 20 facultatifs);
- Toute documentation est autorisée;
- Veuillez à **ne traiter *qu'un* exercice par feuille**, et à **indiquer votre numéro sciper** sur *chacune* des feuilles. Une feuille sans identification ne sera pas corrigée;
- L'examen compte 4 exercices. Ces exercices sont indépendants.
- Les exercices ne sont pas tous de même difficulté. Commencez par ceux dont vous maîtrisez le mieux les concepts impliqués.

SUJET À LA PAGE SUIVANTE

Exercice 1 : Programmation [15 points]

Dans votre premier mini-projet, une méthode

```
public static int[] [] shrink(int[] [] image, int[] seam)
```

vous a été fournie permettant de supprimer un **seam** (chemin de pixels) donné d'une image donnée. Donnez le code d'une méthode

```
public static int[] [] enlarge(int[] [] image, int[] seam)
```

permettant de dupliquer le **seam** donné dans l'image. Le nouveau **seam** sera juxtaposé après celui donné et aura les mêmes valeurs de pixels. L'image obtenue sera d'un pixel plus large.

Pour rappel la convention suivante avait été adoptée : **seam[i]** donne la colonne du pixel dans la ligne **i** de l'image.

Vous considérerez les valeurs des paramètres comme correctes.

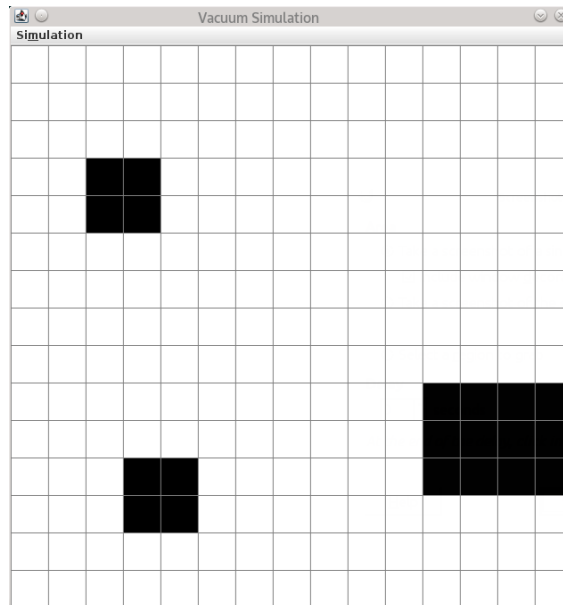
Suite au verso ➞

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

Il est important de lire l'énoncé complètement avant de commencer à répondre.

L'architecture qui vous a été donnée pour mettre en oeuvre votre second mini-projet est donnée en annexe.

Il vous est demandé de l'étendre pour simuler la navigation d'aspirateurs automatiques dans une pièce ressemblant à ceci :



on considérera que :

- La pièce est quadrillée en dalles de carrelage (en blanc, chaque carré est une dalle).
- Toutes les dalles ont une même surface.
- Certaines des dalles sont inaccessibles car couvertes par des meubles (en noir).
- La pièce peut contenir un nombre variable d'aspirateurs.
- Ces derniers se déplacent au hasard de dalle en dalle.
- Un aspirateur a un identificateur (entier) et une capacité d'aspiration (un entier aussi) donnés à la construction. Il est plus petit qu'une dalle et l'aspire quand il est dessus. La dalle devient alors propre.
- Certaines dalles contiennent des petits objets oubliés risquant de se faire aspirer.
- Un petit objet peut être aspiré par un aspirateur si ce dernier n'a pas perdu plus que 50% de sa capacité d'aspiration. Sinon l'aspirateur passe sur la dalle sans l'aspirer et celle-ci reste sale.
- A chaque dalle sale aspirée l'aspirateur perd 3% de sa capacité d'aspiration.
- Une dalle devient propre une fois aspirée et le reste jusqu'à la fin de la simulation.
- Il doit être possible de placer un objet ou un aspirateur à une position donnée.
- L'aspirateur n'a pas connaissance de la pièce dans laquelle il se déplace et n'est pas suffisamment intelligent pour contourner une dalle déjà propre (dans ce cas il n'aspire rien et ne perd pas de capacité d'aspiration).

La simulation doit s'arrêter lorsque toutes les dalles non couvertes par des meubles sont propres ou que tous les aspirateurs ont moins de 10% de leur capacités d'aspiration.

On souhaite alors savoir combien de dalles restent sales et quelles sont les positions des objets oubliés restants (s'il y en a).

La simulation peut être redémarrée. Les aspirateurs recommenceront avec les mêmes position et capacité qu'ils avaient au départ de la précédente simulation. Les objets déjà aspirés ne devront pas réapparaître.

Toutes les positions non occupées par des meubles seront à nouveau considérées comme sales.

Pour simplifier on considérera qu'un petit objet ne chevauche pas deux dalles et que les collisions entre aspirateurs sont négligées.

Toutes les classes utilisées dans le projet, comme `Vector2D` sont à disposition. L'API des classes `World` et `Animal` est donnée en annexe en guise de rappel. Vous considérerez ces classes comme codées et disponibles.

Partie 1 : Conception (36 points)

Il vous est demandé :

1. D'expliciter les membres (attributs et entêtes de méthodes) de toutes les classes nécessaires à l'implémentation du monde physique. Le rôle de chaque classe, membre et méthode sera expliqué au moyen d'un commentaire (si le rôle n'est pas trivial à comprendre).
2. De décrire (en français) quelle classe sera en charge de la simulation, comment elle s'insérera dans l'architecture existante et quel est son rôle.

Pour les deux premiers points, vous pouvez répondre en pseudo-code Java ou au moyen de diagrammes de classes. Le contenu des méthodes n'est pas demandé.

Vous pouvez faire des analogies avec des classes de votre projet pour expliquer le rôle de chaque classe ajoutée.

Contraintes de conception

1. Les constructeurs ne seront pas oubliés ainsi que les éventuelles constantes utiles.
2. Vous porterez une attention particulière aux droits d'accès des attributs et des méthodes et indiquerez clairement ce qui est abstrait, final ou statique dans votre conception.
3. Les méthodes donnant le résultat de la simulation (nombre de dalles restées sales et positions des objets non aspirés) seront codées dans le monde physique.

Partie 2 : Programmation (6 points)

Donnez le code de la méthode permettant de faire le `reset` du monde physique.

Partie 3 : Conception (bonus, 8 points)

La classe `Daedalus` offrait à `DaedalusSimulation` la liste des prédateurs et des proies du monde physique au moyen de getters (`getPredators()` et `getPreys()`).

1. Qu'aurait-il fallu changer au code de `DaedalusSimulation` s'il n'existait qu'un getter `getAnimals()` retournant de façon indistincte les prédateurs et les proies ? Donnez une explication en français.
2. Si ce changement devait se faire sans impliquer de test de type, quel impact cela aurait-il eu sur la hiérarchie des animaux (donnez une explication succincte en français de ce qu'il aurait fallu modifier) ?

Suite au verso ➡

Exercice 3 : Concepts [35 points]

Répondez clairement et succinctement aux questions suivantes :

1. [6 points] Soit le code suivant :

```
0. class A {
1.     protected int a1 = 1;
2.     protected int a2 = 1;
3.     public A(int a1, int a2) {
4.         this.a1=a1;
5.         this.a2=a2;
6.     }
7. }

8. class B extends A {
9.     private int a1 = 1;
10.    private int a2 = 1;
11.    public B(int a1, int a2) {
12.        super(1,1);
13.        this.a1 = a1;
14.        this.a2 = a2;
15.    }
16. }
17. }
```

- (a) Comment remplaceriez vous l'initialisation par défaut de **a1** et **a2** à 1 dans A par un constructeur par défaut ayant une seule instruction ? (donnez le code)
- (b) l'appel à **super(1,1)** à la ligne 13 est-il utile vu que l'on initialise **a1** et **a2** dans les deux instructions suivantes ?
- (c) Comment écririez une méthode dans B affichant sur le terminal la valeur de tous les attributs d'une instance de B. Qu'afficherait cette méthode si on l'invoquait sur une instance de B créée au moyen de **new B(2,3)**
2. [4 points] Soit le programme suivant :

```
class T {
    public static int THRESHOLD = 5;

    public static boolean small(int number) {
        return number < THRESHOLD;
    }

    public static boolean small(int number, int threshold) {
        return number < threshold;
    }
}
```

- (a) la seconde méthode **small** est elle une surcharge de la première. Justifiez brièvement;
- (b) proposez l'entête d'une méthode statique **small** de T qui ne serait pas une surcharge de **boolean small(int number)**.

3. [6 points] Soit la portion de programme suivante :

```
1. class E extends Exception {
2. }

3. class A {
4.     private int a;
5.     public A(int a) throws E {
6.         if (a < 0) {
7.             throw new E();
8.         }
9.         this.a = a;
10.    }
11. }

12. class B extends A {
13.     private int b;
14.
15.     public B(int a, int b) {
16.         super(a);
17.         this.b = b;
18.    }
```

La compilation de ce programme génère le message d'erreur

`unreported exception E; must be caught or declared to be thrown.`

- (a) Expliquez pourquoi.
- (b) Comment rendre ce programme compilable? indiquez quel code ajouter et où.
- (c) Que devrait contenir la classe E au minimum en terme de bonnes pratiques.

4. [5 points] Soit le code suivant :

```
1. interface I {
2.     default void m() {
3.         System.out.println("I::m");
4.     }
5. }

6. class A implements I {
7.     public void m() {
8.         I.m();
9.     }
10. }
```

- (a) Pourquoi ne compile t-il pas ?
- (b) Proposez une correction en ne retouchant qu'à la classe A.
- (c) Est-il possible de déclarer la méthode m de l'interface comme `protected`?
- (d) Est-il possible de déclarer la méthode m de A comme `protected`?

Suite au verso ➞

5. [5 points] Qu'affiche le programme suivant lors de l'exécution de la ligne `A a = new B(2,3);` :

```
abstract class A {
    protected int a;
    public A(int a) {
        this.a = a;
        System.out.println(this);
    }
}
class B extends A {
    private int b;
    public B(int a, int b) {
        super(a);
        this.b = b;
    }
    public String toString() {
        return "L'objet B vaut : " + a + " " + b;
    }
}
```

Justifiez brièvement.

Ce programme fait quelque chose qui n'est pas recommandé dans le constructeur de A. De quoi s'agit-il ? Justifiez brièvement.

6. [6 points] Qu'affiche le programme suivant :

```
1.  class passref
2.  {
3.      public static void main (String[] args) {
4.          String s1 = "est-ce ainsi";
5.          String s2 = "cent hordes sauvages";
6.          String[] s3 = {s2};
7.
8.          f(s1, " meurent ?");
9.          g(s3, " vivent ?");
10.
11.         System.out.println(s1);
12.         System.out.println(s3[0]);
13.     }
14.
15.     private static void f(String s1, String s2)
16.     {
17.         s1 = s1.substring(4, 6);
18.         s1 = s1.replace("e ", "val");
19.         s1 = "que les " + s1;
20.         s1 += s2;
21.     }
22.
23.     private static void g(String[] s1, String s2) {
24.         s1[0] = s1[0].substring(5, 11);
25.         s1[0] = s1[0].replace("rd", "mm");
26.         s1[0] = "que les " + s1[0];
27.         s1[0] += s2;
28.     }
29. }
```

Justifiez brièvement.

Rappels :

- `String.replace(String s1, String s2)` remplace toutes les occurrences de `s1` par `s2` dans `this`;
 - `String.substring(int beginIndex, int endIndex)` extrait de `this` la sous-chaîne commençant à l'indice `beginIndex` (inclus) et se terminant à `endIndex` (exclu).
7. [3 points] Citez une raison importante justifiant l'introduction des méthodes avec définition par défaut dans les interfaces.

SUITE DU SUJET À LA PAGE SUIVANTE

Exercice 4 : Analyse de programme [18 points]

Le propriétaire d'un petit parking a programmé une application pour calculer ce que lui rapportent les véhicules qui y sont parqués.

Les contraintes qu'il a souhaité prendre en compte sont les suivantes :

- Le parking dispose de 10 places pour des voitures et 2 pour des motos.
- Une place de voiture coûte 2 francs par jours, une place de moto 1 franc.
- Un véhicule garé depuis plus de n jours a droit à une réduction (le n et le montant de la réduction sont spécifiques au type de véhicule).

Si toutes les places dédiées aux motos sont prises elles peuvent se garer sur des places voiture. Dans ce cas, le prix de la place reste entier (on paie comme pour une voiture si on occupe une place voiture)

Voici le programme qu'il a produit :

```
0. import java.util.ArrayList;
1. import java.util.List;
2. public class ParkingCorrige {

3.     public static void main(String[] args) {
4.         Parking parking = new Parking();

5.         parking.parquerMoto(new Motorcycle(0), 0);
6.         parking.parquerMoto(new Motorcycle(0), 1);

7.         //pas de places moto vide
8.         parking.parquerVoiture(new Voiture(0), 0);

9.         parking.parquerVoiture(new Motorcycle(0), 1);
10.
11.         System.out.println(parking.calculerPrixTotal(4));

12.     }
13. }
```

Suite au verso ➞

```
14. class Parking {
15.     private Voiture[] placesVoiture = new Voiture[10];
16.     private Motorcycle[] placesMoto = new Motorcycle[2];

17.     public void parquerVoiture(Voiture v, int i) {
18.         if(i >= 0 && i < 10) {
19.             placesVoiture[i] = v;
20.         }
21.     }

22.     public void parquerMoto(Motorcycle m, int i) {
23.         if(i >= 0 && i < 2) {
24.             placesMoto[i] = m;
25.         }
26.     }

27.     private double calculerPrix(Voiture[] vehicules, int jourActuel) {
28.         double somme = 0.0;
29.         for (Voiture v : vehicules) {
30.             if (v != null)
31.                 somme += v.calculerPrix(jourActuel)
32.                     - v.reduction(jourActuel);
33.         }
34.         return somme;
35.     }

36.     public double calculerPrixTotal(int jourActuel) {
37.         double somme = calculerPrix(placesVoiture, jourActuel)
38.             + calculerPrix(placesMoto, jourActuel);
39.         return somme;
40.     }
41. }
```

```
42. class Voiture {
43.     int jourDebut;
44.     public Voiture(int j) {
45.         jourDebut = j;
46.     }

47.     public double calculerPrix(int jourActuel) {
48.         return (jourActuel - jourDebut)* 2;
49.     }

50.     public double reduction(int jourActuel) {
51.         if ((jourActuel - jourDebut) > 15)
52.             return 1.5;
53.         return 0;
54.     }
55. }

56. class Motocycle extends Voiture {

57.     public Motocycle(int j) {
58.         super(j);
59.     }

60.     public double calculerPrix(int jourActuel) {
61.         return super.calculerPrix(jourActuel)/2.;
62.     }

63.     public double reduction(int jourActuel) {
64.         if ((jourActuel - jourDebut) > 20)
65.             return 1.0;
66.         return 0;
67.     }
68. }
```

Questions

1. Expliquez pourquoi le prix total ne sera pas calculé correctement.
2. Quelles critiques pouvez-vous formuler sur sa façon de coder les méthodes `parquerVoiture` et `parquerMoto` (lignes 17 et 22) ?
3. Proposez une conception alternative qui corrige ces problèmes. Cette proposition peut se faire en français et/ou en utilisant du pseudo-code ou des diagrammes de classes.