Examen Programmation Objet - Rattrapage S1 & S2 - 60 pts

Durée : 2 heures Tous documents interdits, Téléphones portables éteints et rangés, barème donné à titre indicatif. Reporter le numéro d'anonymat sur toutes les copies.

Exercice 1 – Vecteur (13 pts)

Soit une classe `Vecteur` et une classe de test associée :

```
1 // Vecteur.java
2 public class Vecteur {
3    public final int id;
4    private static int cpt = 0;
5    public final double x,y;
6    public Vecteur(double x, double y) {
7          id = cpt; cpt++;
8          this.x = x; this.y = y;
9    }
10   public static int getCpt(){return cpt;}
11 }
```

```
12 // TestVecteur.java
13 public class TestVecteur {
14    public static void main(String[] args){
15       Vecteur v1 = new Vecteur(1,2);
16       Vecteur v2 = new Vecteur(1,2);
17       Vecteur v3 = new Vecteur(2,3);
18       Vecteur v4 = v1;
19    }
20 }
```

Q 1.1 (1.5 pt) Expliquer quelles sont les variables d'instance et celles de classe dans la classe `Vecteur`. A la fin de l'exécution du `main`, combien y a-t-il de variables et d'instances en mémoire ? Que vaut `cpt` ?

Q 1.2 (2 pts) Les commandes suivantes compilent-elles (si elles étaient ajoutées à la suite du `main`) ? Pour toutes les commandes qui sont correctes, donner les affichages.

```
21 // suite du main
22    System.out.println(v1.toString()+" "+v2);
23    if(v1==v2) System.out.println("v1 egale v2");
24    if(v1==v3) System.out.println("v1 egale v3");
25    if(v1==v4) System.out.println("v1 egale v4");
26    if(v1==null) System.out.println("v1 egale null");
27    if(v1.equals(v2)) System.out.println("v1 egale v2 (2)");
28    if(v1.equals(v4)) System.out.println("v1 egale v4 (2)");
```

Q 1.3 (4 pts) Ajouter les méthodes suivantes :

- Clonage (méthode clone ou constructeur de copie, au choix). Bien réfléchir à ce qui est souhaitable pour `id` et `cpt`.
- Égalité structurelle standard (`boolean equals(Object o)`),
- Addition : méthode d'addition de vecteurs qui retourne un nouveau vecteur contenant le résultat de l'opération,
- Constructeur sans argument avec initialisation aléatoire des attributs entre 0 (inclus) et 10 (exclus), utilisant obligatoirement l'instruction `this()`.

Q 1.4 (1 pt) A-t-on commis une *faute de conception* en déclarant plusieurs attributs comme public ? (justifier en une phrase)

Q 1.5 (2.5 pts) Les propositions suivantes sont-elles correctes du point de vue syntaxique (compilation) ? Donner les affichages pour les lignes correctes.

```
29 // dans la classe Vecteur
30 public int getCpt2(){return cpt;}
31 public static int getId(){return id;}
32 public static String format(Vecteur v){
33         return String.format("[%5.2f, %5.2f]", v.x, v.y);
34 }
35
36 // dans le main
37 if(v1.x == v2.x && v1.y == v2.y) System.out.println("v1 egale v2");
38 if(v1.id == v2.id) System.out.println("les points ont le meme identifiant");
39 System.out.println("Compteur: "+v1.getCpt());
40 System.out.println("Compteur (2): "+Vecteur.getCpt());
41 System.out.println("Compteur (3): "+v1.cpt);
```

Q 1.6 (2 pts) Remplissage de tableau et références
Dans un `main`, nous avons besoin de remplir un tableau de 100 vecteurs... Afin de faciliter la lecture, ce remplissage est effectué dans la classe `Tools`.

Cas 1 :

```
// main
ArrayList<Vecteur> li = new ArrayList<Vecteur>();
Tools.remplir(li);

// classe Tools
public static void remplir(ArrayList<Vecteur> li){
  for(int i = 0; i<100; i++)
    li.add(new Vecteur(1,0));
}
```

Cas 2 :

```
// main
ArrayList<Vecteur> li = null;
Tools.remplir(li);

// classe Tools
public static void remplir(ArrayList<Vecteur> li){
  li = new ArrayList<Vecteur>();
  for(int i = 0; i<100; i++)
    li.add(new Vecteur(1,0));
}
```

Les deux solutions sont-elles équivalentes ? Dans la négative, laquelle vous semble la plus intéressante ? Justifier brièvement.

Exercice 2 (8 pts) – A, B, ... et exception

```
public class A {
  private int a;
  private static int cpt=0;

  public A(){
    a = cpt; cpt++;
  }
  public int getA(){return a;}
  public static int getCpt(){return cpt;}
  public void ajout1(){a++;}
  public String toString(){
    return "A"+a;
  }
}
```

```
public class B{
  private A a1, a2;

  public B(A a1, A a2){
   this.a1 = a1; this.a2 = a2;
  }
  public B(){
    this(new A(), new A());
  }
  public void ajout1(){
    a1.ajout1(); a2.ajout1();
  }
}
```

Q 2.1 (1 pt) Dans le programme suivant, donner les affichages.

```
A a1 = new A(); A a2 = new A(); A a3 = new A();
A a4 = a1; A a5 = a2; A a6 = a3;
a5.ajout1();
System.out.println(a1+"␣"+a2+"␣"+a3);
System.out.println(a1.getCpt()+"␣"+a1.getA()+"␣"+A.getCpt()));
```

Q 2.2 (2.5 pts) Soit la suite du programme, donner le nombre d'instances de A et B à la fin de l'exécution ainsi que le diagramme de la mémoire.

```
B b1 = new B(); B b2 = new B(a1, a2); B b3 = new B(new A(), a3); B b4 = b1;
a6.ajout1();
b2.ajout1();
```

Q 2.3 (0.5 pt) Donner les affichages suite à l'exécution du code suivant :

```
// suite du programme precedent
System.out.println(a2 == a1);
System.out.println(a2 == a5);
System.out.println(a2 == a6);
```

Q 2.4 (4 pts) Donner **tout** le code nécessaire pour implémenter le test d'égalité structurel dans la classe B.

Exercice 3 (≈ 16 pts) – Évolution architecturale

Dans le cadre d'un jeu vidéo, on souhaite avoir des objets qui évoluent au cours du temps, au fur et à mesure que l'on passe des niveaux du jeu par exemple.
Pour ce faire, nous disposons d'une classe très simple `UpgradableObject` qui est un conteneur d'objet pouvant évoluer :

```
1 public class UpgradableObject{
2         public Upgradable o;
3         public UpgradableObject(Upgradable o){ this.o = o;}
4         public void upgrade(){
5             o=o.upgrade();
6         }
7 }
```

Q 3.1 (1 pt) Supposons que l'on dispose d'un tableau d'objets `UpgradableObject[] tab`, donner les instructions pour faire évoluer tous les objets de ce tableau.

Q 3.2 (1.5 pt) Le type d'objet `Upgradable` est en fait défini par une interface. Donner le code de l'interface `Upgradable`.
Note : attention au type que doit retourner la méthode `upgrade` des objets de type `Upgradable`.

Q 3.3 (1 pt) Soit une classe **abstraite** `Batiment` implémentant l'interface `Upgradable`. Cette classe est-elle soumise à une contrainte particulière du fait d'implémenter `Upgradable` ?

Q 3.4 (1.5 pt) Donner le code de `Batiment`, sachant que tous les bâtiments possèdent des coordonnées GPS et une surface (initialisées par le constructeur).

Les bâtiments peuvent être soit des bâtiments de niveau 1, des bâtiments de niveau 2 et des bâtiments de niveau 3. Les bâtiments de niveau 1 permettent uniquement d'héberger 2 personnes. Lors du passage au niveau 2, la capacité d'hébergement est doublée. Lors du passage au niveau 3, le bâtiment est capable de stocker également 2 objets `Marchandise`.
Note : on dispose des classes `Personne` et `Marchandise`.

Q 3.5 (1 pt) Donner un diagramme de l'architecture générale du programme (incluant l'ensemble des classes de l'exercice)

Q 3.6 (7 pts) Donner le code des trois classes de bâtiment `Batiment1`, `Batiment2` et `Batiment3`, en respectant les spécifications et le mécanisme d'évolution des bâtiments des questions précédentes.

- Pour gagner du temps, ne donner que les signatures des accesseurs et modifieurs.
- Bien réfléchir à la méthode `upgrade` et aux constructeurs nécessaires à chaque niveaux.

Q 3.7 (1 pt) Que pensez-vous du programme principal suivant ?

```
1 public static void main(String[] args){
2         Batiment[] tab=new Batiment[3];
3         tab[0]=new Batiment1(10, 12, 125); // constructeur existant
4         tab[1]=new Batiment2(12, 13, 150); // constructeur existant
5         tab[2]=new Batiment1(5, 3, 100);
6         for(int i=0;i<3;i++){
7                 tab[i].upgrade();
8         }
9 }
```

Q 3.8 (3 pts) Donner un programme principal de test qui crée trois instances de `Batiment1`, les range dans un tableau, fait un upgrade sur tous les éléments du tableau et qui permette en sortie d'avoir un tableau de trois `Batiment2` **en utilisant les mécanismes/structures prévus dans l'exercice**.
Ajouter une fonction qui teste la nature du premier élément du tableau afin de vérifier que l'upgrade a bien eu lieu.

Problème : la machine à café (28 pts)

On considère une machine à café. A la base du système, nous allons d'abord considérer les **ingrédients** : le **café**, l'**eau**, le **lait**, le **chocolat**. La **machine à café** est ensuite constituée de **réservoirs** pour les **ingrédients** et de **recettes** (expresso, café allongé, café au lait, chocolat chaud...). Afin d'éviter de recharger la machine en eau trop souvent, elle sera reliée à un **robinet** qui sera modélisé comme un réservoir de capacité infinie.

Exercice 4 – Éléments de base (7 pts)

Q 4.1 (2.5 pts) Hiérarchie(s) de classes. Situer tous les éléments en gras de la description dans une arborescence de classes en utilisant les liens d'héritage et de composition. Indiquer la/les classe(s) abstraite(s).

Q 4.2 (1 pt) Donner le code de la classe abstraite `Ingredient` contenant un attribut `String nom`, un constructeur à un argument pour initialiser ce nom, un méthode `toString`.
Dans la suite de l'exercice, le nom servira à décrire l'ingrédient (e.g. `"café"`, `"chocolat"`...)

Q 4.3 (2 pts) Ajouter une méthode standard `equals` testant l'égalité structurelle entre 2 ingrédients (c'est à dire entre leurs `noms`) en traitant le cas où `nom` n'est pas instancié (`null`).
Note : attention à bien considérer la classe `String` comme un objet et pas un type de base.

Q 4.4 (0.5 pt) Une classe abstraite a-t-elle forcément une méthode abstraite ?

Q 4.5 (1 pt) Donner le code de la classe `Cafe` héritant d'`Ingredient`. Le constructeur ne prend pas d'argument : le `nom` étant toujours `cafe`.
On imagine dans la suite du problème que tous les ingrédients (classes `Eau`, `Lait` et `Chocolat`) ont également été créés avec un constructeur sans paramètre.

Exercice 5 – Reservoir (8 pts)

Q 5.1 (1.5 pt) Donner le code d'une classe `RecuperationIngredientException` qui étend les exceptions et qui a un constructeur à un argument (`String message`).

Q 5.2 (1.5 pt) Le réservoir a pour attribut un `Ingredient`, une `capacite` (un réel exprimé en litre), un `niveau` (également réel exprimé en litre). A la construction le réservoir est plein. Le réservoir dispose d'un accesseur sur l'ingrédient et d'une méthode `remplir` qui le remplit totalement.
Donner le code de la classe.

Q 5.3 (3 pts) Le réservoir a aussi une méthode `recuperer` qui prend en argument un réel (la quantité souhaitée par le client) : la méthode teste si le `niveau` est suffisant et, dans l'affirmative, décrémente le `niveau`, sinon elle lève une `RecuperationIngredientException` avec un message expliquant le problème (type d'ingrédient et quantité disponible). Une fois sur mille (aléatoirement), le réservoir connait une défaillance et n'est pas capable de délivrer l'ingrédient : vous lèverez une exception dans ce cas, également avec un message explicite.
Donner le code de la méthode en portant une attention particulière à la signature.

Q 5.4 (2 pts) Donner le code de la classe `Robinet`, qui est un réservoir de capacité infinie (`Double.POSITIVE_INFINITY`) qui rencontre un problème quand on s'en sert une fois sur 500 (aléatoirement). Le message à donner à l'utilisateur est de vérifier que le robinet est bien ouvert.

Exercice 6 – Recette (2 pts)

Une recette est composée d'un tableau d'ingrédients de taille fixe et d'un tableau de réels indiquant les quantités (toujours en litre). Elle a aussi un prix (réel, en euros) et un nom. Le constructeur prend en arguments tous les éléments nécessaires à l'initialisation des attributs. La classe possède des accesseurs sur tous ses champs.

Q 6.1 (2 pts) Donner le code de la classe : attributs + constructeur (aucun piège ici, donner rapidement le code). On considère que les accesseurs existent et s'appellent `getNomAttribut()` (pas la peine de donner le code pour gagner du temps).

Exercice 7 – Machine (12 pts)

La machine est composée de deux listes dynamiques (`ArrayList`[1]) de recettes et de reservoirs. La machine gère un crédit (réel, en euros) et elle a un identifiant entier unique que vous initialiserez avec un compteur static.
De manière générale, la machine fonctionne de la manière suivante. D'abord, l'utilisateur ajoute de l'argent à la machine, puis sélectionne une recette, enfin la machine prépare la mixture en récupérant chaque ingrédient dans son réservoir. La machine sait rendre la monnaie (dans la pratique, on mettra simplement le crédit disponible à 0).

Q 7.1 (2 pt) Donner le code de base de la classe `Machine` avec un constructeur sans argument et deux méthodes `public void ajouterReservoir(Reservoir r)` et `public void ajouterRecette(Recette r)` pour configurer la machine. La classe possède aussi une méthode `public void ajouterCredit(double d)` pour mettre de l'argent (en euros) et une méthode `public void rendreLaMonnaie()` qui met le crédit à 0.

Q 7.2 (1 pt) Donner le code de la méthode `public void remplir()` qui correspond à l'action de l'agent d'entretien consistant à remplir tous les réservoirs au maximum.

Q 7.3 (2 pts) Nous allons maintenant procéder au *check-up* de la machine pour vérifier que tout est OK. Nous avons besoin d'une méthode `private Reservoir trouverReservoir(Ingredient i)` qui retourne le réservoir associé à l'ingrédient `i` s'il existe dans la machine et `null` sinon. On suppose qu'il n'y a qu'un réservoir par ingrédient pour éviter les complications.
Donner le code de cette méthode et expliquer en une phrase pourquoi nous l'avons déclaré `private`.

Q 7.4 (2 pts) Donner le code de la méthode `public boolean checkup()` qui vérifie si toutes les recettes sont réalisables, c'est à dire, si tous les ingrédients de toutes les recettes sont bien disponibles dans la machine (à la première erreur, le test est interrompu et retourne `false`). Cette méthode affiche dans la console le nom des recettes avec la mention OK à coté si la recette est réalisable. Elle retourne `true` si toutes les recettes sont OK.

Q 7.5 (5 pts) Donner le code de la méthode `public boolean commander(int ri)` correspondant à l'appui sur le bouton `ri` de la machine. La machine indique la recette sélectionnée (si elle existe), vérifie que l'utilisateur a assez de crédit et prépare la recette en affichant des messages au fur et à mesure de la préparation. En cas de problème lors de la préparation, la machine affiche un message et retourne `false`. Le client n'est débité que si tout s'est bien passé (dans ce cas, on retourne `true`).
Note : on considère ici que le *check-up* a été passé avec succès, tous les réservoirs associés à une recette sont toujours disponibles (ils peuvent simplement être vides ou en panne).

Exercice 8 – Test (4 pts)

Q 8.1 (4 pts) Donner le code du programme `TestMachineACafe` permettant de valider le bon fonctionnement de la machine : création d'une recette simple à 2 ingrédients (et ajout dans la machine), création des 2 réservoirs associés (et ajout dans la machine), check-up, test sur la monnaie disponible, test pour distribuer des boissons jusqu'à provoquer une erreur.

Rappel de documentation : `ArrayList<Object>`

Il faut ajouter la commande `import java.util.ArrayList;` en début de fichier pour utiliser les `ArrayList`.

- Instanciation : `ArrayList<Object> a = new ArrayList<Object>();`
- `void add(Object o)` : ajouter un élément à la fin
- `Object get(int i)` : accesseur à l'item `i`
- `Object remove(int i)` : retirer l'élément et renvoyer l'élément à la position `i`
- `int size()` : retourner la taille de la liste
- `boolean contains(Object o)` retourne `true` si `o` existe dans la liste. Le test d'existence est réalisé en utilisant `equals` de `o`.

1. Mini-documentation disponible en fin d'énoncé.