

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`?

0.5 pt chaque

- Instance : `id`, `x`, `y`
- Classe : `cpt`

-0.5 pt par faute

4 variables, 3 instances
`cpt = 3`

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)");
```

- 0.5pt = Aucune erreur
- 0.5 : `System.out.println(v1.toString()+" "+v2);`
- 1 pt pour le résultat des égalités (pas de double punition s'ils ont noté des erreurs)

```
1 exam.exam2015c.Vecteur@74a14482 exam.exam2015c.Vecteur@1540e19d
2 v1 egale v4
3 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()`.

```

— 0.5 : clonage
— 0.5 : constructeur
— 1 : addition
— 2 : equals (-1 si pas de test sur null, -1 si pas de test avant le cast, -2 si pas de cast du tout ou mauvaise signature)

1  public Vecteur addition(Vecteur a){
2      return new Vecteur(x+a.x, y+a.y);
3  }
4  public boolean equals(Object obj) {
5      if (this == obj) // OPTION
6          return true;
7      if (obj == null)
8          return false;
9      if (getClass() != obj.getClass())
10         return false;
11     Vecteur other = (Vecteur) obj;
12     if (x != other.x || y != other.y)
13         return false;
14     return true;
15 }
16 public Vecteur clone(){
17     return new Vecteur(x, y);
18 }
19 public Vecteur(){
20     this(Math.random()*10,Math.random()*10);
21 }

```

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

Tous les attributs public sont final : ils sont donc protégés (non modifiables depuis le client). Pas de problème.

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_egal_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);

```

-0.5pt par faute / affichage manquant ou incorrect

```

29 // dans la classe Vecteur
30 public int getCpt2(){return cpt;} // OK
31 public static int getId(){return id;} // NON: static => pas d'accès aux attributs
32 public static String format(Vecteur v){ // OK
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_egal_v2"); // OK: v1 egale v2
38 if(v1.id == v2.id) System.out.println("les_points_ont_le_meme_identifiant"); // OK
39 System.out.println("Compteur: "+v1.getCpt()); // OK : Compteur: 3
40 System.out.println("Compteur(2): "+Vecteur.getCpt()); // Compteur (2): 3
41 System.out.println("Compteur(3): "+v1.cpt); // NON: champs prive
```

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 :

```
1 // main
2 ArrayList<Vecteur> li = new ArrayList<Vecteur>();
3 Tools.remplir(li);
4
5 // classe Tools
6 public static void remplir(ArrayList<Vecteur> li){
7     for(int i = 0; i<100; i++){
8         li.add(new Vecteur(1,0));
9     }
```

Cas 2 :

```
1 // main
2 ArrayList<Vecteur> li = null;
3 Tools.remplir(li);
4
5 // classe Tools
6 public static void remplir(ArrayList<Vecteur> li){
7     li = new ArrayList<Vecteur>();
8     for(int i = 0; i<100; i++){
9         li.add(new Vecteur(1,0));
10 }
```

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

2 pts

Gauche = OK Droite = n'importe quoi...

C'est pas pareil

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

```
1 public class A {
2     private int a;
3     private static int cpt=0;
4
5     public A(){
6         a = cpt; cpt++;
7     }
8     public int getA(){return a;}
9     public static int getCpt(){return cpt;}
10    public void ajout1(){a++;}
11    public String toString(){
12        return "A"+a;
13    }
14 }
```

```
1 public class B{
2     private A a1, a2;
3
4     public B(A a1, A a2){
5         this.a1 = a1; this.a2 = a2;
6     }
7     public B(){
8         this(new A(), new A());
9     }
10    public void ajout1(){
11        a1.ajout1(); a2.ajout1();
12    }
13 }
```

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

```
1 A a1 = new A(); A a2 = new A(); A a3 = new A();
2 A a4 = a1; A a5 = a2; A a6 = a3;
3 a5.ajout1();
4 System.out.println(a1+" "+a2+" "+a3);
5 System.out.println(a1.getCpt()+" "+a1.getA()+" "+a1.getCpt());
```

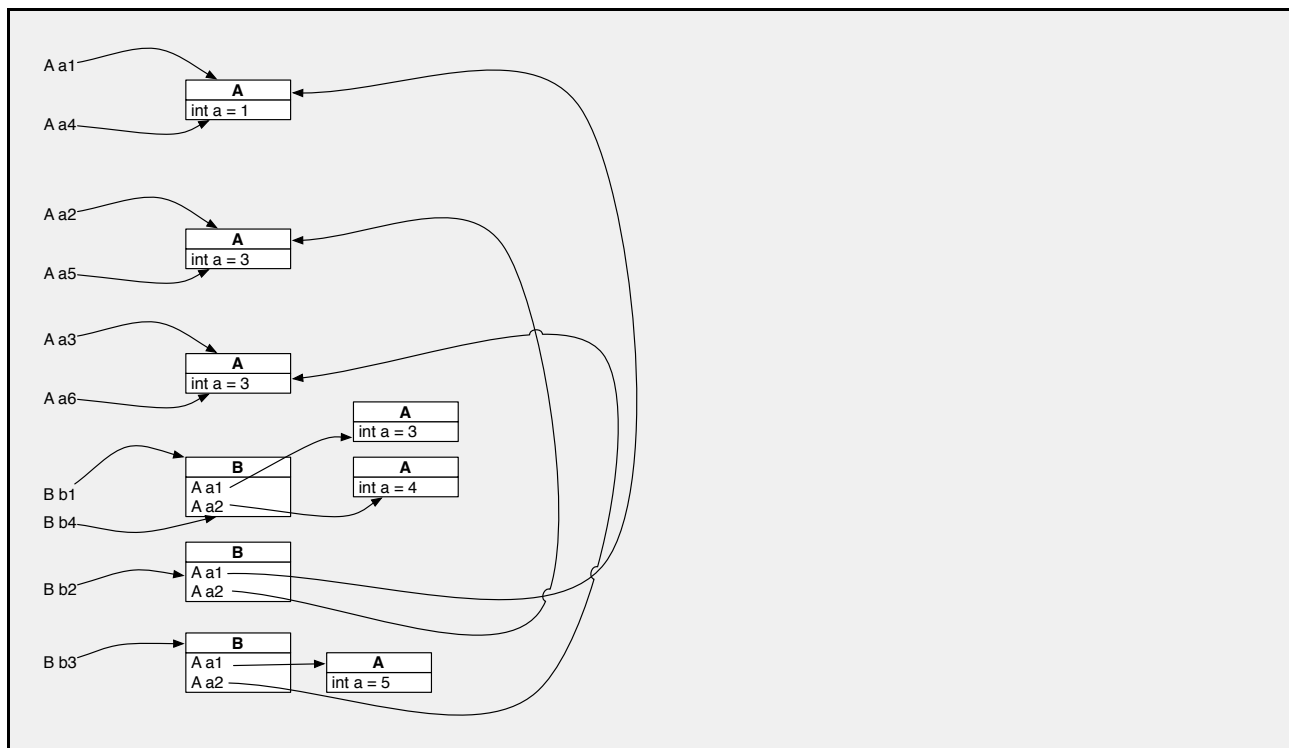
A0 A2 A2

3 0 3

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.

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

Il y a 6 instances de A et 3 de B



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

```

9 // suite du programme precedent
10 System.out.println(a2 == a1);
11 System.out.println(a2 == a5);
12 System.out.println(a2 == a6);

```

false
true
false

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

Equals dans A et 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.

```

1 pt
1 // dans un main
2     for(int i=0;i<tab.length;i++){
3         tab[i].upgrade();
4     }

```

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`.

```

1.5 pt
1     public interface Upgradable{
2         public Upgradable upgrade();
3     }

```

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` ?

```

1 pt
Non, c'est une classe abstraite, pas besoin d'implémenter upgrade

```

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).

```

1.5 pts = 1pt signature + 0.5 reste
1     public abstract class Batiment implements Upgradable{
2         private String lat;
3         private String lon;
4         private double surface;
5         public Batiment(String lat, String lon, double sur){
6             this.lat=lat;
7             this.lon=lon;
8             this.surface=sur;
9         }
10    }

```

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.

Batiment1 = 3.5 pts Batiment2 = 2 pts Batiment3 = 1.5 pt

```

1 public class Batiment1 extends Batiment {
2     private Personne p1, p2;
3
4     public Batiment1(double lat, double lon, double sur) {
5         super(lat, lon, sur);
6     }
7
8     public Upgradable upgrade() {
9         return new Batiment2(this);
10    }
11
12    public Personne getP1() {return p1;}
13    public void setP1(Personne p1) {this.p1 = p1;}
14    public Personne getP2() {return p2;}
15    public void setP2(Personne p2) {this.p2 = p2;}
16 }
17
18 public class Batiment2 extends Batiment1{
19     private Personne p3, p4;
20
21     protected Batiment2(double lat, double lon, double sur) {
22         super(lat, lon, sur);
23     }
24     public Batiment2(Batiment1 b){
25         super(b.getLat(), b.getLon(), b.getSurface());
26     }
27
28     @Override
29     public Upgradable upgrade() {
30         return new Batiment3(this);
31     }
32
33     public Personne getP3() {return p3;}
34     public void setP3(Personne p3) {this.p3 = p3;}
35     public Personne getP4() {return p4;}
36     public void setP4(Personne p4) {this.p4 = p4;}
37 }
38
39 public class Batiment3 extends Batiment2 {
40     private Marchandise m1, m2;
41
42     public Batiment3(Batiment2 b){
43         super(b.getLat(), b.getLon(), b.getSurface());
44     }
45
46     @Override
47     public Upgradable upgrade() {
48         return new Batiment3(this);
49     }
50
51     public Marchandise getM1() {return m1;}
52     public void setM1(Marchandise m1) {this.m1 = m1;}
53     public Marchandise getM2() {return m2;}

```

```

54     public void setM2(Marchandise m2) {this.m2 = m2;}
55 }

```

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 }

```

1 pt

upgrade ne change rien au tableau

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.

3 pts

```

1     public static void main(String[] args){
2         UpgradableObject[] tab=new UpgradableObject[3];
3         tab[0]=new UpgradableObject(new Batiment1(...));
4         tab[1]=new UpgradableObject(new Batiment1(...));
5         tab[2]=new UpgradableObject(new Batiment1(...));
6         for(int i=0;i<3;i++){
7             tab[i].upgrade();
8         }
9
10        if(tab[0].o instanceof Batiment2)
11            System.out.println("upgrade OK");
12    }

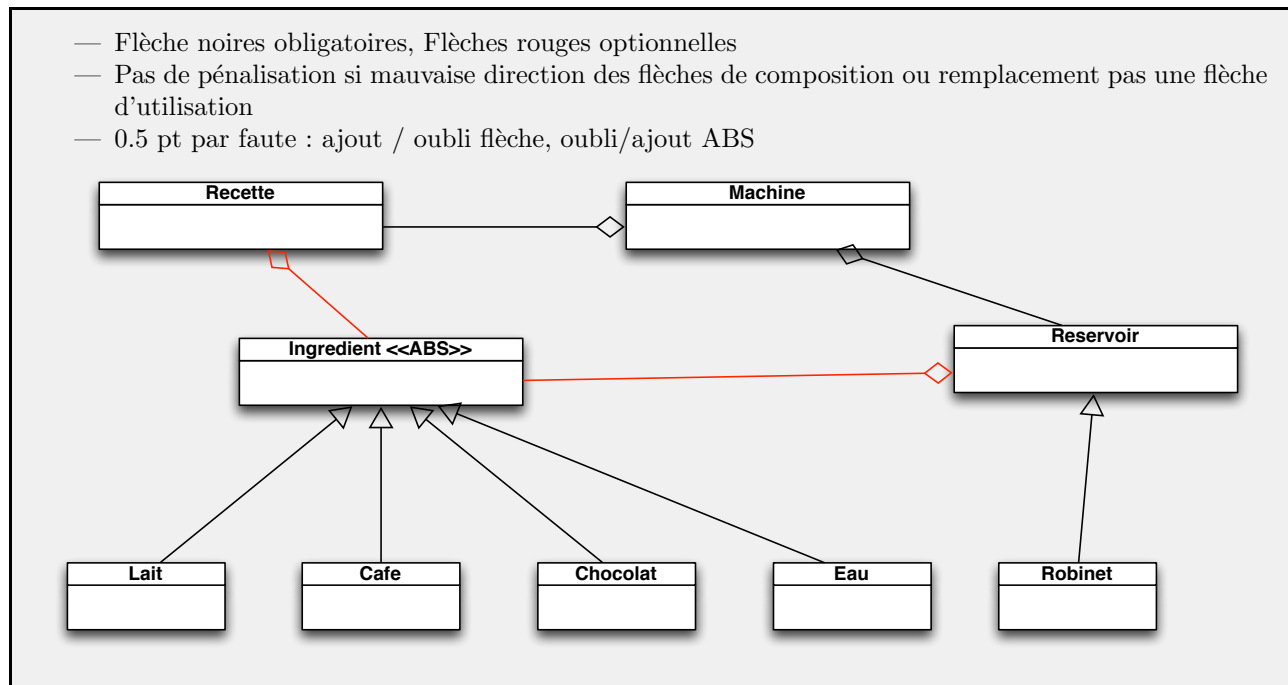
```


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"...)

— 1 pt signature class + attribut + constructeur
 — 2 pts equals : 1 pt pour tout + 1 pt pour le double test de nullité

```

1 public abstract class Ingredient {
2     private String nom;
3
4     public Ingredient(String nom) {
5         this.nom = nom;
6     }
7
8     public String toString(){
9         return nom;
10    }
11
12    public boolean equals(Object obj) {
13        if (this == obj)
14            return true;
15        if (obj == null)
16            return false;
17        if (getClass() != obj.getClass())
18            return false;
  
```

```
19     Ingredient other = (Ingredient) obj;
20     if (nom == null) {
21         if (other.nom != null)
22             return false;
23     } else if (!nom.equals(other.nom))
24         return false;
25     return true;
26 }
27 }
```

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 ?

Non

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`.

```
1 public class Cafe extends Ingredient{
2     public Cafe() {
3         super("cafe");
4     }
5 }
```

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`).

```
1 public class RecuperationIngredientException extends Exception {
2     public RecuperationIngredientException(String message) {
3         super(message);
4     }
5 }
```

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.

```
1
2 //=====
3 //1.5 pt
4
5 public class Reservoir {
6     private Ingredient ingredient;
7     private double niveau;
```

```

8   private double capacite;
9
10  public Reservoir(Ingredient ingredient, double capacite) {
11      this.ingredient = ingredient;
12      this.niveau = capacite;
13      this.capacite = capacite;
14  }
15
16  public Ingredient getIngredient() {
17      return ingredient;
18  }
19
20  public void remplir() {
21      niveau = capacite;
22  }
23
24  //=====
25  //3 pts
26
27  public void recuperer(double qte) throws RecuperationIngredientException { // 1 pt pour
    la signature
28  // 1.5 pt pour les 2 tests
29      if(Math.random() < 0.001)
30          throw new RecuperationIngredientException("Probleme_avec_le_reservoir_de_"+
    ingredient);
31      if(qte > niveau)
32          throw new RecuperationIngredientException("Reservoir_de_"+ingredient+"_vide");
33      // 0.5 pour le code
34      niveau -= qte;
35      return;
36  }
37
38 }

```

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 connaît 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.

```

1 public class Robinet extends Reservoir {
2     public Robinet(Ingredient ingredient) {
3         super(ingredient, Double.POSITIVE_INFINITY);
4     }
5
6     public void recuperer(double qte) throws RecuperationIngredientException {
7         if(Math.random() < 1./500)
8             throw new RecuperationIngredientException("Probleme_avec_le_robinet_de_"+
    getIngredient()+"_verifier_que_le_robinet_est_ouvert");
9
10        return;
11    }
12 }

```

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).

Pas de difficulté ici : barème de la dictée, entre -0.5pt et -1 pt par faute (selon la gravité).

```

1 public class Recette {
2     private Ingredient[] elements;
3     private double[] quantites;
4     private double prix;
5     private String nom;
6
7     public Recette(Ingredient[] elements, double[] quantites,
8         double prix, String nom) {
9         this.elements = elements;
10        this.quantites = quantites;
11        this.prix = prix;
12        this.nom = nom;
13    }
14    // ===== Fin du code exige dans la question =====
15    // NON demande aux etudiants
16    public Ingredient[] getElements() {
17        return elements;
18    }
19
20    public double[] getQuantites() {
21        return quantites;
22    }
23
24    public double getPrix() {
25        return prix;
26    }
27
28    public String getNom() {
29        return nom;
30    }
31 }

```

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.

Pour les reservoir et la monnaie, vérifier la compréhension générale

```

1 import java.util.ArrayList; // -0.5 si oubli de declaration
2
3 public class Machine {
4     private ArrayList<Recette> recettes;
5     private ArrayList<Reservoir> reservoirs;
6     private double credit;
7     private static int cpt = 0; // -0.5 si mauvaise gestion du compteur

```

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

```

8     private int id;
9
10    public Machine(){ // -1 pt si oubli d'instanciation des arraylist
11        recettes = new ArrayList<Recette>();
12        reservoirs = new ArrayList<Reservoir>();
13        id = cpt++;
14    }
15
16    public void ajouterReservoir(Reservoir r){
17        reservoirs.add(r);
18    }
19
20    public void ajouterRecette(Recette r){
21        recettes.add(r);
22    }
23    public void rendreLaMonnaie(){
24        credit = 0;
25    }
26    // En EURO
27    public void ajouterCredit(double d){
28        credit += d;
29    }

```

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.

```

1 public void remplir(){
2     for (Reservoir r:reservoirs)
3         r.remplir();
4 }

```

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`.

0.5 pour l'explication : `private` car n'a pas vocation à être appelée depuis le main (plus simple à lire pour le client, plus sécurisé).

```

1 private Reservoir trouverReservoir(Ingredient i){
2     for(Reservoir res:reservoirs){
3         if(res.getIngredient().equals(i)){ // -1 si pas de equals
4             return res;
5         }
6     }
7     return null;
8 }

```

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 côté si la recette est réalisable. Elle retourne `true` si toutes les recettes sont OK.

```

1 public boolean checkup(){
2     System.out.println("Les recettes disponibles sont :");
3     for(Recette r:recettes){
4         System.out.print(r.getNom()); // pas de retour pour pouvoir afficher OK a cote

```

```

5         for(Ingredient i: r.getElements()){
6             if(trouverReservoir(i) == null)
7                 return false;
8         }
9         System.out.println("OK");
10    }
11
12    System.out.println("Machine_"+id+"_tout_est_OK.");
13    return true;
14 }

```

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).

- Vérifier les tests : 1 pt
- Vérifier la bonne gestion des exception (pas de throws, bon try catch..., return dans le catch pour ne pas débiter) : 2 pts
- Fonctionnement général : 2pt

```

1  public boolean commander(int ri){
2      if(ri>=recettes.size()){
3          System.out.println("recette_non_disponible");
4          return false;
5      }
6      System.out.println("recette_: "+recettes.get(ri).getNom());
7      if(credit<recettes.get(ri).getPrix()){
8          System.out.println("credit_insuffisant");
9          return false;
10     }
11
12     try{
13         for (int i=0; i<recettes.get(ri).getElements().length; i++){
14             trouverReservoir(recettes.get(ri).getElements()[i]).recuperer(recettes.get(ri).getQuantites()[i]);
15             System.out.println("Machine_: j'ai verse_ "+recettes.get(ri).getQuantites()[i]+" de_ "+recettes.get(ri).getElements()[i]);
16         }
17     }catch(RecuperationIngredientException e){
18         System.out.println(e.getMessage());
19         return false;
20     }
21     credit -= recettes.get(ri).getPrix();
22     System.out.println("livraison OK!");
23     return true;
24 }

```

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.

```

1 public static void main(String[] args) {

```

```

2      Machine machineACafe = new Machine();
3
4      machineACafe.ajouterReservoir(new Reservoir(new Cafe(), 2));
5      machineACafe.ajouterReservoir(new Robinet(new Eau()));
6      // OPT
7      //machineACafe.ajouterReservoir(new Reservoir(new Lait(), 2));
8
9      machineACafe.ajouterRecette(new Recette(new Ingredient[] {new Cafe(), new Eau()},
10         new double[] {0.02,0.1}, 0.35, "expresso"));
11     // OPT
12     //machineACafe.ajouterRecette(new Recette(new Ingredient[] {new Cafe(), new Lait()
13         , new Eau()},new double[] {0.02,0.05, 0.1}, 0.35, "cafe au lait"));
14     //machineACafe.ajouterRecette(new Recette(new Ingredient[] {new Cafe(), new Eau()
15         },new double[] {0.02,0.15}, 0.35, "cafe allonge"));
16
17     System.out.println("check_up: "+machineACafe.checkup());
18
19     machineACafe.remplir();
20
21     machineACafe.commander(0);
22
23     machineACafe.chargerMonnaie(1000);
24
25     boolean b = machineACafe.commander(0);
26
27     while(b)
28         b=machineACafe.commander(0);
29 }

```

Rappel de documentation : ArrayList<Object>

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

- `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`.