Examen Programmation Objet (60 pts)

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

Questions de cours (27 pts)

Exercice 1 – Vecteur (11 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 pt) Expliquer quelles sont les variables d'instance et celles de classe dans la classe `Vecteur`.

0.5 pt chaque
- Instance : `id, x, y`
- Classe : cpt

Q 1.2 (1 pt) 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 par faute
4 variables, 3 instances
cpt = 3

Q 1.3 (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 1.4 (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 1.5 (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 1.6 (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);
```

-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'acces 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 egale 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
```

Exercice 2 – Héritage (8.5 pts)

```
1 public abstract class Fourmi{
2     protected String nom;
3     public Fourmi(String nom){this.nom = nom;}
4     public void manger(Nourriture n){
5         System.out.println(nom+" mange "+n);
6     }
7 }
8
9 public class Ouvriere extends Fourmi{
10    public Ouvriere(String nom){super(nom);}
11 }
12
13 public class Reine extends Fourmi{
14    private int cpt;
15    public Reine(String nom){
16        super(nom); cpt=0;
17    }
18    public void manger(GeleeRoyale g){
19        System.out.println(nom+
20                           " (Reine) mange de "+g);
21    }
22    public Fourmi engendrer(){
23        cpt ++;
24        return new Ouvriere(nom+cpt);
25    }
26 }
```

```
28 public class Nourriture{
29     private String description;
30     public Nourriture(String description){
31       this.description = description;
32     }
33     public String toString(){
34       return description;
35     }
36 }
37
38 public class GeleeRoyale extends Nourriture{
39     public GeleeRoyale(){
40       super("gelee pour la reine");
41     }
42 }
```

Q 2.1 (3.5 pts) Vrai/Faux général sur l'héritage. Parmi les instructions suivantes, identifier celles qui sont incorrectes et expliquer succinctement le problème (en précisant s'il survient au niveau de la compilation ou de l'exécution). Donner le nom des fourmis qui ont effectivement été engendrées par une reine.

```
43 Fourmi f1    = new Fourmi("f1");
44 Fourmi f2    = new Ouvriere("ouv1");
45 Ouvriere f3 = new Ouvriere("ouv2");
46 Fourmi f4    = new Reine("majeste1");
47 Ouvriere f5 = new Reine("majeste2");
48 Reine f6     = new Reine("majeste3");
49 Fourmi[] fourmilliere = new Fourmi[100];
50 f2.manger(new Nourriture("sucre"));
51 fourmilliere[0] = f4.engendrer();
52 fourmilliere[1] = f5.engendrer();
53 fourmilliere[2] = f6.engendrer();
54 fourmilliere[3] = ((Reine) f2).engendrer();
55 fourmilliere[4] = ((Reine) f4).engendrer();
56 fourmilliere[5] = ((Reine) f6).engendrer();
```

-0.5 par faute (non détection d'err, ajout d'err, faute dans le nom de la fourmi)

```
43 Fourmi f1  = new Fourmi("f1"); // KO compil: classe abs => pas de new
44 Fourmi f2  = new Ouvriere("ouv1");
45 Ouvriere f3 = new Ouvriere("ouv2");
46 Fourmi f4 =  new Reine("majeste1");
47 Ouvriere f5 =  new Reine("majeste2"); // KO compil: incoherence dans la subsomption
48 Reine f6 =  new Reine("majeste3");
49 Fourmi[] fourmilliere = new Fourmi[100];
50 f2.manger(new Nourriture("sucre"));
51 fourmilliere[0] = f4.engendrer(); // KO compil:  methode non visible sur une variable
       Fourmi
```

```
fourmilliere[1] = f5.engendrer(); // KO : variable non creee // pas trop de penalisation:
      seulement si l'etudiant est manifestement incoherent par rapport a la declaration.
fourmilliere[2] = f6.engendrer(); // OK : majeste31
fourmilliere[3]=((Reine) f2).engendrer(); // KO execution : ClassCastException
fourmilliere[4]=((Reine) f4).engendrer(); // OK : majeste11
fourmilliere[5]=((Reine) f6).engendrer(); // OK : majeste32
```

Q 2.2 (2 pt) Sélection de méthodes. Donner les affichages lors de l'exécution du code suivant :

```
Reine r1 =  new Reine("majeste1");
Fourmi r2 =  new Reine("majeste2");
r1.manger(new Nourriture("un peu de sucre"));
r1.manger(new GeleeRoyale());
r2.manger(new Nourriture("un peu de viande"));
r2.manger(new GeleeRoyale());
```

- 1 pt pour les 3 faciles
- 1 pt pour `majeste2 mange gelee pour la reine`

Piege sur r2 sur la gelée royale :

```
majeste1 mange un peu de sucre
majeste1 (Reine) mange de gelee royale pour la reine
majeste2 mange un peu de viande
majeste2 mange gelee pour la reine
```

Q 2.3 (3 pts) Donner les changements et/ou ajouts de code pour que la reine refuse de manger autre chose que de la gelée royale (on peut lui en donner mais elle affiche un message de refus dans la console).
Note : vous opterez pour la solution la plus élégante.

- 1.5 pt s'ils ont vu qu'il fallait faire une redefinition propre au lieu de la *spécialisation* qui marche mal
- 1.5 pt pour le reste du code

```
// remplacer public void manger(GeleeRoyale g) par :

    public void manger(Nourriture n){
        if(!(n instanceof GeleeRoyale)){
            System.out.println("Je refuse de manger !");
            return;
        }
        System.out.println(nom+" (Reine) mange de la gelee");
    }
```

Exercice 3 – Programme mystère (7.5 pts)

```java
public class Obj {
    private int i,j;
    public Obj(int i, int j) {
        this.i = i;
        this.j = j;
    }
    public void maj(Obj o){
        System.out.println("maj");
        i += o.i;
    }
    public int getResult(){
       System.out.println("res");
        return i+j;
    }
}
```

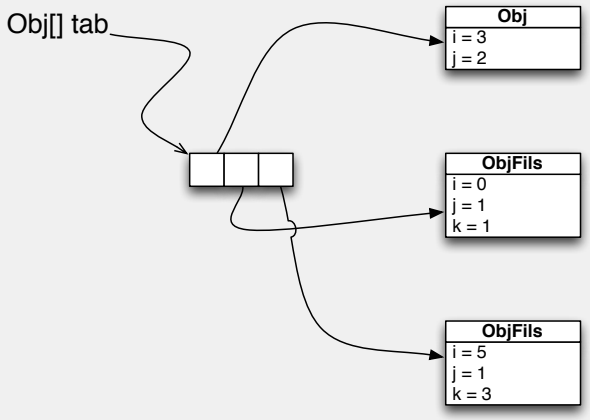
```java
public class ObjFils extends Obj{
    private int k;
    public ObjFils(int i, int j) {
        super(i, j);
        this.k = i+j+(int) Math.random();
    }
    public ObjFils(int m) {
        this(m, m+1);
    }
    public int getResult(){
       System.out.println("res (fils)");
        return k+4;
    }
}
```

```java
public class Mystere {
    public static void main(String[] args) {
        Obj[] tab = {new Obj(1,2), new ObjFils(0),new ObjFils(2,1)};
        for(int i=0; i<tab.length; i++) tab[i].maj(tab[tab.length-1-i]);
        int s = 0;
        for(Obj o:tab) s+=o.getResult();

        System.out.println(s);
    }
}
```

Q 3.1 (1 pt) Dessiner un diagramme mémoire correspondant à la fin de l'exécution du code sans préciser les valeurs numériques des attributs.

Assez trivial sans les valeurs numériques... Mais je veux voir les types de la variable et des instances.



Q 3.2 (0.5 pts) Que penser du code `(int) Math.random()` ?

0.5 pt : `(int) Math.random()` vaut toujours 0

Q 3.3 (2.5 pts) Donner les affichages produits par ce programme.

1 pt : affichage des méthodes (respect du nb d'appel à `maj` + sélection des méthodes `res`)
1.5 pt : pour le calcul de 17

```
maj
```

```
2 maj
3 maj
4 res
5 res (fils)
6 res (fils)
7 17
```

Q 3.4 (3.5 pts) On propose d'ajouter le code suivant dans la classe `ObjFils` :

```
1 // Classe ObjFils
2     public void maj(ObjFils of){
3         System.out.println("maj (fils)");
4         super.maj(of);
5         k += of.k;
6     }
```

Q 3.4.1 (0.5 pt) Est-il nécessaire d'inverser les lignes 3 et 4 pour permettre la compilation ?

Non, ne pas confondre `super.` et `super()`

Q 3.4.2 (1.5 pt) A la fin de l'exécution du code suivant, que valent les attributs `i,j,k` de l'objet `of` ?

```
1 ObjFils of = new ObjFils(1);
2 of.maj(new Obj(1,1));
3 of.maj(new ObjFils(1,1));
```

0 si faux, 1.5 si juste :)

```
1 ObjFils of = new ObjFils(1); // 1, 2, 3
2 of.maj(new Obj(1,1)); // 2, 2, 3
3 of.maj(new ObjFils(1,1)); // Resultat attendu : 3, 2, 5
```

Q 3.4.3 (0.5 pt) Que se passe-t-il si on oublie `super.` à la ligne 4 ? (Réponse 1 : pas d'impact, réponse 2 : ça change tout -dans ce cas expliquer-)

Ca donne une récursion infinie (plantage sur StackOverFlow)

Q 3.4.4 (1 pt) L'ajout de cette méthode modifie-t-il les affichages du programme précédent ? Dans l'affirmative, donner les nouveaux affichages (sauf pour la ligne 37, pas besoin de refaire les calculs).

Non, aucun changement, la méthode n'est jamais sélectionnée !

Rappel de documentation :

Dans une `ArrayList<Object>`, les méthodes suivantes sont utiles :

- 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

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

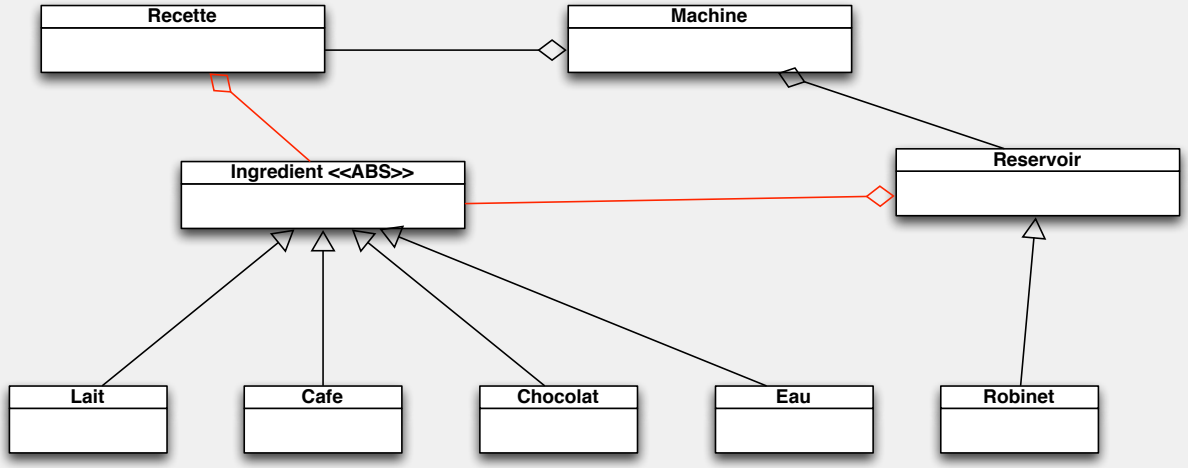
Problème : la machine à café (33 pts)

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).

- Flèche noires obligatoires, Flèches rouges optionnelles
- Pas de pénalisation si mauvaise direction des flèches de composition ou remplacement pas une flèche d'utilisation
- 0.5 pt par faute : ajout / oubli flèche, oubli/ajout ABS



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 //1.5pt
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 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.

```
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;
```

1. Mini-documentation disponible avant le problème.

```
5     private ArrayList<Reservoir> reservoirs;
6     private double credit;
7     private static int cpt =0; // -0.5 si mauvaise gestion du compteur
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 a 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 coté 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("livrason␣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.

```java
public static void main(String[] args) {
        Machine machineACafe = new Machine();

        machineACafe.ajouterReservoir(new Reservoir(new Cafe(), 2));
        machineACafe.ajouterReservoir(new Robinet(new Eau()));
        // OPT
        //machineACafe.ajouterReservoir(new Reservoir(new Lait(), 2));

        machineACafe.ajouterRecette(new Recette(new Ingredient[] {new Cafe(), new Eau()},
            new double[] {0.02,0.1}, 0.35, "expresso"));
        // OPT
        //machineACafe.ajouterRecette(new Recette(new Ingredient[] {new Cafe(), new Lait()
            , new Eau()},new double[] {0.02,0.05, 0.1}, 0.35, "cafe au lait"));
        //machineACafe.ajouterRecette(new Recette(new Ingredient[] {new Cafe(), new Eau()
            },new double[] {0.02,0.15}, 0.35, "cafe allonge"));

        System.out.println("check up: "+machineACafe.checkup());

        machineACafe.remplir();

        machineACafe.commander(0);

        machineACafe.chargerMonnaie(1000);

        boolean b = machineACafe.commander(0);

        while(b)
            b=machineACafe.commander(0);
    }
```

Exercice 9 – Extensions bonus

Q 9.1 (3 pts) Pour créer une machine connectée, on invente un nouveau type de `ReservoirConnecte` qui est construit avec 3 arguments `(Ingredient ingredient, double capacite, String adresse)`. Il a une méthode `void mailTo(String message)` permettant d'envoyer un mail à la compagnie qui gère la machine (en utilisant l'`adresse` donnée lors de la construction). La machine envoie automatiquement un mail lorsqu'un réservoir passe à un niveau de remplissage inférieur à 10% (dans la pratique, on affiche simplement un message dans la console).
Donner le code de cette classe.
Note : on supposera qu'il existe des accesseurs pour les attributs `niveau` et `capacite` dans la classe `Reservoir`.

```java
// dans resservoir
    public double getPcRemplissage(){
        return niveau/capacite;
    }

public class ReservoirConnecte extends Reservoir{
    private String adresse;

    public ReservoirConnecte(Ingredient ingredient, double capacite, String adresse) {
        super(ingredient, capacite);
        this.adresse = adresse;
    }

    private void mailTo(String message){
        System.out.println(message);
    }

    public void recuperer(double qte) throws RecuperationIngredientException{
        if(getPcRemplissage()<0.1)
            mailTo("Mail a: "+adresse+" : reservoir de "+getIngredient()+" presque vide");
        super.recuperer(qte);
    }
```

```
23 }
```

Q 9.2 (2 pts) L'architecture de la machine est-elle satisfaisante et évolutive ? Justifier en expliquant la démarche pour ajouter un programme soupe à la tomate.

— 1 pt Oui, c'est une bonne architecture : on peut ajouter des fonctionnalités sans modifier le code existant, simplement en ajoutant des classes
— 1 pt Ajouter la classe Tomate, Ajouter un reservoir de Tomate + Ajouter une recette avec Tomate + Eau.