Examen Programmation Objet S1 - 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 (5 pts) – Introduction

```
1 // tous les imports nécessaires sont présents
2 public class TestDessinBase {
3   public static void main(String[] args){
4     BufferedImage im = new BufferedImage(300, 300, BufferedImage.TYPE_4BYTE_ABGR);
5     String filename = "monimage.png";
6     ImageIO.write(im, "png", new File(filename));
7   }
8 }
```

Q 1.1 (1.5 pt) Pour chacun des éléments suivants, indiquer s'il s'agit d'une variable `static`, d'une constante (`final static`), d'une variable d'instance ou d'une variable locale, d'une méthode `static` ou d'une méthode d'instance :

- `TYPE_4BYTE_ABGR`
- `im`
- `write`

0.5 pt par réponse
- Constante
- variable locale
- méthode static

Q 1.2 (2.5 pts) La sauvegarde de l'image est susceptible de lever une exception de type `IOException`. Le programme peut-il compiler en l'état ? Dans tous les cas, proposer 2 solutions distinctes pour améliorer la gestion de cette potentielle exception.

0.5 pt NON, ça ne compile pas
1 pt - Solution 1 :

```
1 try{
2   ImageIO.write(im, "png", new File(filename));
3 } catch(IOException e){
4   e.printStackTrace();
5 }
```

1 pt - Solution 2

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

On compte juste si les étudiants ont bien décrit la solution, donner les bons mots clés, même s'il n'y a pas de code.

Q 1.3 (1 pt) Comment faire pour sauver l'image dans un fichier dont le nom est donné lors du lancement du programme ? Donner la modification de code à effectuer et la commande à lancer dans la console.

0.5 pt

```
1 ImageIO.write(im, "png", new File(filename)); // devient
2 ImageIO.write(im, "png", new File(args[0]));
```

0.5 pt - Commande :

```
1 java TestDessinBase nomfichier
```

Exercice 2 (17 pts) – Comptage d'instances, connaissances de base

```
1 public class Point{
2     private double x,y;
3     public Point(double x, double y) {
4         this.x = x;
5         this.y = y;
6     }
7     public Point addition(Point p){
8         return new Point(x + p.x, y+p.y);
9     }
10    public String toString() {
11        return "Point [x="+x+", y="+y+"]";
12    }
13 }
```

```
1 public class Test {
2   public static void main(String[] args){
3     Point p = new Point(1,1);
4     Point[] tab = new Point[10];
5     for(int i=0; i<6; i++)
6       tab[i] = p.addition(new Point(i, 1));
7     System.out.println(p);
8     Point p2 = p;
9     p2 = p.addition(tab[0]);
10    System.out.println(p+" "+p2);
11  }
12 }
```

Q 2.1 (1.5 pt) Combien d'instances de `Point` sont créées au total lors de l'exécution du programme ? Combien d'instances existent encore à la ligne 10 du `main` ?

0.75 pt
1 + 6*2 + 1 = 14 instances
0.75 pt
1 + 1 + 6 = 8 instances (les autres sont déréférencées et passées au GC)

Q 2.2 (1.5 pt) Qu'est ce qui s'affiche lors de l'exécution du programme ?

évidemment, p n'est pas modifié... Mais les étudiants ont tendance à s'emmêler entre instance et référence...
-0.5 s'il manque le premier affichage
-0.5 s'ils ne sont pas capable d'avoir une sortie qui ressemble à la forme du toString.
-1 si la deuxième ligne incorrecte

```
1 Point [x=1.0, y=1.0]
2 Point [x=1.0, y=1.0] Point [x=2.0, y=3.0]
```

Q 2.3 (1.5 pt) Nous ajoutons maintenant le code suivant dans le `main` :

```
11    for(int i=0; i<tab.length; i++)
12        System.out.println(tab[i].toString());
```

Donner l'ensemble des affichages liés à l'exécution de ces 2 lignes.

0.75 pour les valeurs (pour les 6 instances)
0.75 pour l'exception

```
1 Point [x=1.0, y=2.0]
2 Point [x=2.0, y=2.0]
3 Point [x=3.0, y=2.0]
4 Point [x=4.0, y=2.0]
5 Point [x=5.0, y=2.0]
6 Point [x=6.0, y=2.0]
7 Exception in thread "main" java.lang.NullPointerException
8         at cours1.TestPoint.main(TestPoint.java:24)
```

Q 2.4 (1 pt) Dans `Point`, donner le code de la méthode de clonage `public Point clone()`.

```
1 public Point clone(){return new Point(x, y);}
```

Q 2.5 (1 pt) Donner le code de base de la classe `PointNomme` qui étend `Point` et ajoute un attribut de type `String` gérant le `nom` du `Point`. [Signature de la classe, déclaration de l'attribut, constructeur]

```java
public class PointNomme extends Point { // 0.5 pt signature + attribut
    private String name;

    public PointNomme(double x, double y, String nom) {
        super(x, y); // 0.5 pt usage de super
        this.nom = nom;
    }
```

Q 2.6 (3 pts) Les lignes suivantes sont-elles correctes (toujours à la suite dans le `main`) pour le compilateur ? Pour la JVM ? Donner la sortie console associée aux deux dernières lignes.

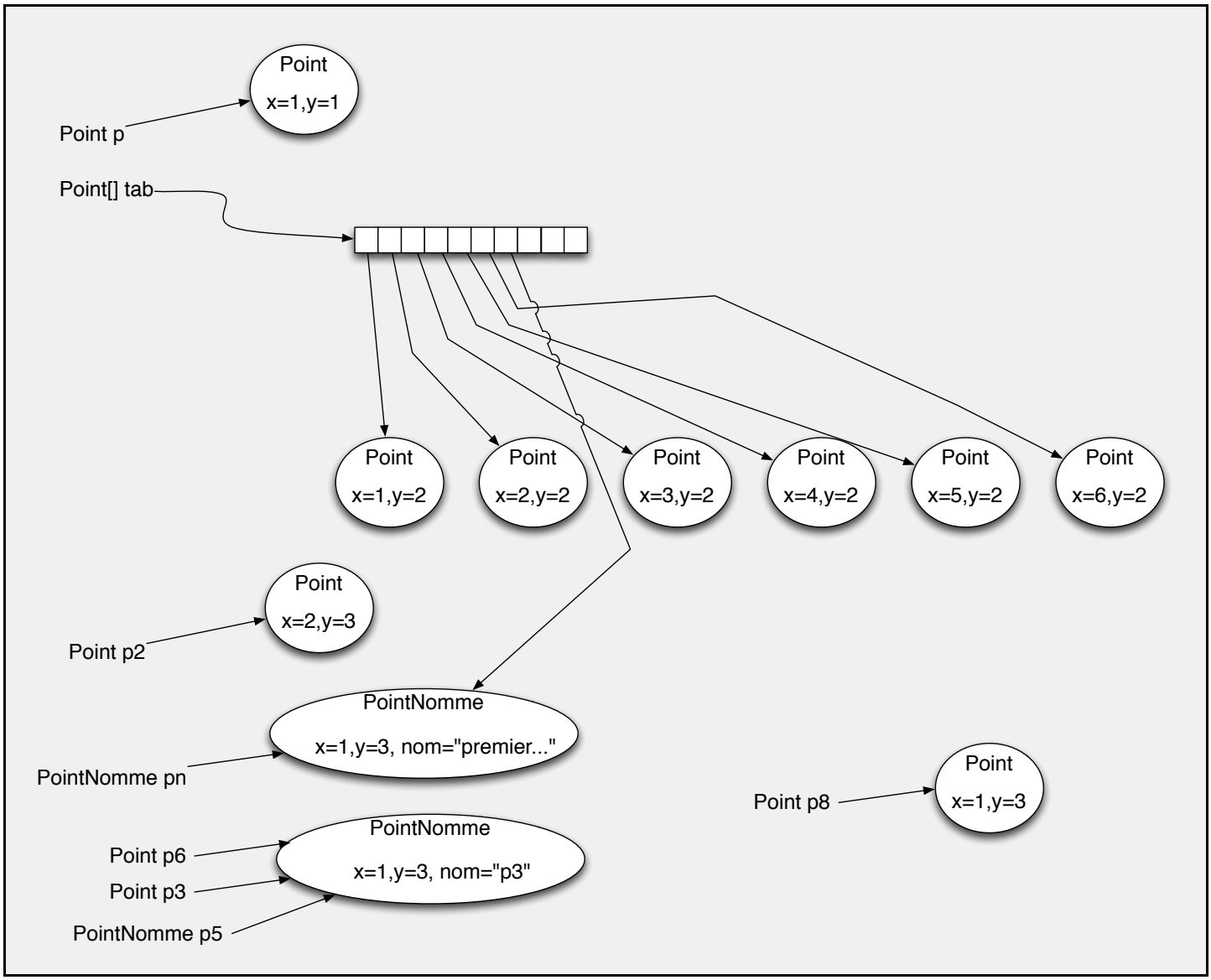
```java
PointNomme pn = new PointNomme(1, 3, "premier point nommé");
Point p3 = new PointNomme(1, 3, "p3");
PointNomme p4 = new Point(1, 3);
PointNomme p5 = (PointNomme) p3;
Point p6 = (Point) p5;
PointNomme p7 = (PointNomme) p;
Point p8 = pn.clone();
PointNomme p9 = pn.clone();
tab[6] = pn;
System.out.println(pn);
System.out.println(tab[7]);
```

-1 par faute ou non réponse
-0.5 pour les petites fautes (affichage...)
Il faut être sympa dans la correction... Trop de fautes constatées en général... On ne retire que 0.25 s'il y a une faute mais des choses intéressantes qd même

```java
PointNomme pn = new PointNomme(1, 3, "premier point nommé"); // OK
Point p3 = new PointNomme(1, 3, "p3"); // OK: subsombtion
PointNomme p4 = new Point(1, 3); // NON : compil KO
PointNomme p5 = (PointNomme) p3; // OK
Point p6 = (Point) p5; // OK (mais cast inutile)
PointNomme p7 = (PointNomme) p; // NON. compil OK mais JVM ClassCastException
Point p8 = pn.clone(); // OK
PointNomme p9 = pn.clone(); // NON (clone renvoie un Point)
tab[6] = pn; // OK
System.out.println(pn); // OK: Point [x=1, y=3]
System.out.println(tab[7]); // OK: null
```

Q 2.7 (3 pts) Donner le diagramme de l'état de la mémoire à l'issue de l'exécution de l'ensemble du `main` (sauf les lignes posant problème). Pour gagner de la place, vous ne mentionnerez pas les instances détruites par le garbage collector.

On ne compte pas les fautes des exo précédent, on se concentre sur les déclarations de variables et les types des instances. On n'hésite pas à mettre 0 si ça ne ressemble à rien et qu'il manque des types de variable ou d'instance... J'ai passé pas mal de temps sur ces représentations en cours ! Et en plus, j'ai garanti qu'ils en auraient à l'examen.



Q 2.8 Nous souhaitons maintenant contraindre les points à avoir des coordonnées entre 0 et 10, en levant une exception `CoordinateOutOfBoundException` en cas de non respect.

Q 2.8.1 (1.5 pt) Donner le code de la classe `CoordinateOutOfBoundException` qui permet de construire une exception avec un **réel en argument**. Lorsque l'exception est levée, elle enverra le message :
`Coordonnée non comprise entre 0 et 10: xxx`, où xxx est la valeur aberrante constatée.

0.75 pour les signatures de classe et de constructeur
0.75 pour la gestion du super et du message.

```
1 public class CoordinateOutOfBoundException extends Exception {
2     public CoordinateOutOfBoundException(double x) {
3         super("coordonnée non comprise entre 0 et 10: "+x);
4     }
5 }
```

Q 2.8.2 (1.5 pts) Donner la nouvelle version du constructeur de `Point`.

-1 si mauvaise syntaxe throw
-1 si oubli de throws

```java
public Point(double x, double y) throws CoordinateOutOfBoundException {
        this.x = x;
        this.y = y;

        if(x<0 || x> 10)
            throw new CoordinateOutOfBoundException(x);
        if(y<0 || y> 10)
            throw new CoordinateOutOfBoundException(y);
}
```

Q 2.8.3 (1 pt) Le constructeur de `PointNomme` est-il impacté par notre modification ? Dans l'affirmative, que faut-il modifier ?

Oui, il faut déclarer `throws CoordinateOutOfBoundException` à cause de l'usage de `super`

Q 2.8.4 (0.5 pt) D'autres méthodes posent-elles problème ? [Pas de code demandé]

0.25pt si ils ne citent qu'une méthode
Oui, addition, clone... Toutes celles qui utilisent le constructeur.

Exercice 3 (14 pts) – Quelques notes de musique

Nous proposons dans cet exercice de gérer une partition de musique. Pour gagner du temps, nous repartons de classes existantes comme `Note` qui modélise une gamme grave de piano et la possibilité de transposer les notes dans les gammes au dessus (pour info : l'opération correspond à une multiplication de la fréquence).

```java
public final class Note {
    // chaque note correspond à une fréquence
    public static final Note do_ = new Note(65.4064);
    public static final Note re_ = new Note(73.4162);
    public static final Note mi_ = new Note(82.4069);
    public static final Note fa_ = new Note(87.3071);
    public static final Note sol_ = new Note(97.9989);
    public static final Note la_ = new Note(110);
    public static final Note si_ = new Note(123.471);
    public static final Note silence_ = new Note(0);
    // coefficient multiplicateur pour les demi ton (dièse/bémol)
    private static final double demiTon = 1.05946;

    public final double frequence;

    private Note(double frequence) { this.frequence = frequence; }

    // pour les générer les demis tons
    public Note diese(){ return new Note(frequence*demiTon); }
    public Note bemol(){ return new Note(frequence/demiTon); }
    // pour passer dans une gamme au dessus (facteur = nb de gamme au dessus)
    public Note transpose(int facteur){ return new Note(Math.pow(2., facteur)*frequence); }
}
```

Q 3.1 (3 pts) Parmi les opérations suivantes, toutes effectuées en dehors de la classe `Note`, lesquelles posent problème ? Expliquer très brièvement.

```java
Note n1 = new Note(220);
Note n2 = Note.do_;
Note n3 = n2.re_;
Note si_bemol = Note.si_.bemol;
System.out.println(n2.frequence);
Note.mi_.frequence = 12;
Note do_aigu = Note.do_.transpose(1);
Note do_diese = Note.do_.diese();
Note do_diese2 = Note.do_ * Note.demiTon;
Note do_aigu2 = Note.do_.transpose(2.5);
```

-0.75 par faute, non justification ou oubli de faute
-0.5 pour les petites fautes (bemol l4, détection d'un pb l3)

```
1 Note n1 = new Note(220); // impossible (constr private)
2 Note n2 = Note.do_; // OK
3 Note n3 = n2.re_; // OK
4 Note si_bemol = Note.si_.bemol; // NON: appel de méthode sans parenthèses
5 System.out.println(n2.frequence); // OK
6 Note.mi_.frequence = 12; // NON: attribut final
7 Note do_aigu = Note.do_.transpose(1); // OK
8 Note do_diese = Note.do_.diese(); // OK
9 Note do_diese2 = Note.do_ * Note.demiTon; // NON : attribut privé
10 Note do_aigu2 = Note.do_.transpose(2.5); // NON : int attendu, pas double
```

Q 3.2 (1 pt) Quels sont les points forts/faibles de l'architecture de cette classe ?

+ Bien sécurisée : pas de possibilité de créer des notes absurdes
+ accès direct à la frequence + sécurisation car cst
- pas vraiment de point faible... Obligation de créer une nouvelle instance pour une nouvelle note... Mais ce n'est pas vraiment un point faible.

Q 3.3 (0.5 pt) Pour ajouter une notion de rythme (noire, croche, blanche...), nous allons développer une nouvelle classe abstraite `Rythme` et des classes filles concrètes. Est-il possible de faire hériter `Rythme` de `Note` ?

Non (classe final) - Obligation pour l'étudiant de mentionner `final`

Q 3.4 (2.5 pts) Donner le code de la classe `Rythme`, qui gère une note et sa durée (`double`) et possède deux accesseurs vers la durée et la fréquence. Donner aussi le code de la classe `Noire` qui dure `1.0` temps.

```
1 public abstract class Rythme { // 0.5 abstract + attribut private
2     private double duree;
3     private Note n;
4
5     public Rythme(double duree, Note n) {
6         this.duree = duree;
7         this.n = n;
8     }
9     public double getDuree(){
10        return duree;
11    }
12    public double getFreq(){ // 0.5 pour les accésseurs (accès direct à la frequence obligatoire)
13        return n.frequence;
14    }
15 }
16 public class Noire extends Rythme { // 0.5 extends
17    public Noire( Note n) { // 1 constructeur
18        super(1., n);
19    }
20 }
```

Q 3.5 (1.5 pt) Donner le code de la classe `Partition` qui étend la classe `ArrayList<Rythme>` et qui gère un attribut `double tempo` (pour indiquer à quelle vitesse jouer cette partition). Cette classe doit (évidemment) gérer l'ajout et la récupération de notes. Elle possède aussi un accesseur pour le `tempo`. [Doc. de `ArrayList` en bas de page]

Pdt l'examen, bcp de question sur la *récupération* de notes... J'aurais du mettre une formule plus détaillée du style : accéder à la i^e notes du tableau. Il faut être sympa avec ceux qui ce sont égarés !

Par contre : sévère si mauvais attributs, si redéfinition de méthode alors qu'elles sont héritées... Encore pire, s'ils héritent de `Rythme` ou autre bêtise du genre.
Note : j'ai prévenu en cours que je demanderais probablement d'étendre une `ArrayList`. Donc, pas de tolérance sur cette difficulté

```
// -1 si ajout de méthode existante
// -1 si ajout d'attribut
public class Partition extends ArrayList<Rythme>{
    private double tempo;

    public Partition(double tempo) {
        super(); // facultatif
        this.tempo = tempo;
    }

    public double getTempo() {
        return tempo;
    }
}
```

Q 3.6 (3.5 pts) En cherchant sur internet, nous avons trouvé une classe permettant de générer des sons : `Player`... Cette classe n'est pas compatible avec notre architecture : elle attend pour sa construction un argument de type `Iterator<double[]>`. Un iterateur est une interface qui impose l'implémentation des méthodes suivantes :

```
public interface Iterator<double[]>{ // (version simplifiée pour éviter les génériques)
  public boolean hasNext(); // existe-t-il un élément suivant?
  public double[] next(); // retourne l'élément suivant
}
```

Chaque élément `double[]` correspondra à un triplet contenant le temps de départ du son (en seconde), sa durée (en seconde) et sa fréquence.
Donner le code de la classe `Traducteur`, qui répond à la spécification `Iterator<double[]>` et qui prend en argument une `Partition`.
Note : le `Traducteur` doit gérer le défilement du temps en secondes. Au début, le temps est à 0 ; à chaque fois que nous récupérons une note dans la partition, le compteur est incrémenté de : $duree_{note} * tempo_{partition}/60$. De la même manière, la durée d'une note en seconde vaut : $duree_{note} * tempo_{partition}/60$.

```
public class Traducteur implements Iterator<double[]>{ // 1pt pour la signature

    private Partition p;
    private int index;
    private double t;
    public Traducteur(Partition p) {
        this.p = p;
        index = 0;
        t = 0;
    }

    public boolean hasNext() { // 1 pt (notamment pour le bon accès à la longueur de la partition)
        return index < p.size();
    }
    public double[] next() { // 1.5 pt pour la méthode + attributs cohérents et bonne gestion du
        temps
        Rythme r = p.get(index);
        index ++;

        double[] retour = new double[]{t, r.getDuree()* p.getTempo()/60., r.getFreq()};
        t += r.getDuree() * p.getTempo()/60.;
        return retour;
    }
}
```

Q 3.7 (2 pts) Proposer une classe de test qui construit une partition de 3 notes, la donne à un traducteur puis vérifie le bon fonctionnement de celui-ci en faisant défiler les triplets et en les affichant dans la console.

```
    public static void main(String[] args)  {
        Partition p = new Partition(60);
        p.add(new Noire(Note.do_));
        p.add(new Noire(Note.re_));
        p.add(new Noire(Note.mi_));

        Traducteur t = new Traducteur(p);
        while(t.hasNext()){
            double[] triplet = t.next();
            System.out.println(triplet[0]+" "+triplet[1]+" "+triplet[2] );
        }
    }
```

Exercice 4 (7 pts) – **Mauvaise idée...**

```
public class A {
   private int i;
   private String str;
   public A(int i, String str){
       this.i = i;
       this.str = str;
   }
}
```

```
public class Test {
 public static void main(String[] args){
   A a1 = new A(Integer.valueOf(args[0]), args[1]);
   A a2 = new A(1, "titi");
   Object[] tab = {5,6.2,"toto",new A(1, "titi"),a1,a2};
   for(Object o: tab){
      System.out.println(o+" : "+a2.equals(o)+" "+o.equals(a2));
   }
   System.out.println(a1.equals(a2));
 }
}
```

Q 4.1 (0.5 pt) Etant donné qu'il n'y a pas de problème avec la fonction `valueOf` ni avec les exceptions : le programme compile-t-il ? Sinon, proposer une correction rapide.

Pas de problème... Il existe une méthode `equals` dans `Object`

Q 4.2 (1.5 pt) Nous souhaitons l'exécuter avec la ligne suivante :

```
java Test 1 titi
```

Considérons que le programme s'exécute maintenant correctement (sans exception). Qu'est ce qui s'affiche dans la console ?

1 pour les booleens
0.5 pour le toString par défaut de `A`

```
5 : false false
6.2 : false false
toto : false false
interro.exam2016quest.A@15db9742 : false false
interro.exam2016quest.A@6d06d69c : false false
interro.exam2016quest.A@7852e922 : true true
false
```

Q 4.3 (3 pts) Afin d'améliorer le comportement du programme, le développeur propose une (très mauvaise) solution : il ajoute la ligne suivante dans la classe *A*.

```
public boolean equals(A a){  return i == a.i && str == a.str; }
```

Que deviennent les sorties console ? Expliquer les résultats des comparaisons entre instances de `A` en insistant sur le choix de la méthode `equals`.

On regarde uniquement les booléens. 0 si pas d'explication.
(1) = sélection de la méthode `equals(Object)` dans la classe `Object`
(2) = sélection de la méthode `equals(A)` dans la classe `A`

```
5 : false false
6.2 : false false
toto : false false
// 0.5 pt - a2.equals(o) : (1) on ne regarde que le type de la variable argument => (1) + pas d'égalité ref. = false
// 1 pt - o.equals(a2) : (1) Mécanisme complexe! Compil = sélection de equals(Object) dans Object; execution, sélection de equals(Object) dans A... Qui existe par héritage de Object! => (1) + pas d'égalité ref. = false
interro.exam2016quest.A@15db9742 : false false
// 0 pt - idem ci-dessus
interro.exam2016quest.A@6d06d69c : false false
// 0.25 pt - a2.equals(o) : égalité référentielle dans (1), true
// 0.25 pt - o.equals(a2) : égalité référentielle dans (1), true
interro.exam2016quest.A@7852e922 : true true
// 1 pt car pas d'égalité référentielle entre String !! (dit et redit en cours)
false
```

Q 4.4 (2.5 pts) Donner le code qu'il aurait du proposer pour obtenir un comportement satisfaisant (pour simplifier, nous considérons que les `String` ne sont jamais `null`). Attention à bien distinguer les tests entre types de bases et ceux entre objets.

Redefinition **exacte** de la méthode `equals`.

```
public boolean equals(Object obj) { // -2 pts  si signature fausse
        if (this == obj)          return true;
        if (obj == null)          return false;
        if (getClass() != obj.getClass())       return false;
        A other = (A) obj; // -1 pt s'il manque le cast ou le test précédent
        if (i != other.i)        return false;
        if (str == null) { // OPT             if (other.str != null)
                return false;
        } else if (!str.equals(other.str)) // 1pt pour .equals entre String
            return false;
        return true;
```

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`.

Exercice 5 (18 pts) – Recettes de cuisine

Soit le programme principal suivant, garanti sans erreur d'aucune sorte, qui permet de simuler la recette du gâteau au yaourt. L'idée est d'utiliser une hiérarchie de classes pour créer différents ingrédients, les mélanger, les cuire... De manière à transformer une recette de cuisine en algorithme.

[Rappel de syntaxe] : `new Object[]{4., "toto", ...}` permet de créer une instance de tableau rapidement.

```
public class TestRecette {
    public static void main(String[] args){
        Preparation oeufs  = new Ingredient("oeufs", 3);
        Preparation yaourt = new Ingredient("yaourt", 1, "pot");
        Preparation farine = new Ingredient("farine", 3, "pots de yaourt");
        Preparation sucre  = new Ingredient("sucre", 2, "pots de yaourt");
        Preparation huile  = new Ingredient("huile", 0.5, "pot de yaourt");
        Preparation levure = new Ingredient("levure", 1, "paquet");

        Operation melange  = new AjoutMelange(new Preparation[]{yaourt, farine, sucre});
        Operation melange2 = new AjoutMelange(new Preparation[]{melange, oeufs});
```

```
12          Operation melange3 = new AjoutMelange(new Preparation[]{melange2, huile, levure});
13          Operation cuisson  =
14                new CuissonAuFour(new Preparation[]{melange3}, "(Préchauffer le four)", 180);
15          Preparation gateauAuYaourt =
16                new Recette(new Preparation[]{cuisson}, "Gateau au yaourt");
17
18          // Exemple d'opération qui doivent fonctionner
19          System.out.println(oeufs.getNom());
20          PreparationComposite groupeDePreparations = melange;
21          Preparation[] tableau = groupeDePreparations.getTableau();
22          for(Preparation p: tableau)
23              System.out.println(p.getNom());
24          Ingredient i = (Ingredient) yaourt;
25          System.out.println(i.getNom()); // yaourt
26          System.out.println(i.getQuantite()); // 1.
27          System.out.println(i.getUnite()); // pot
28      }
29  }
```

Q 5.1 (2 pts) Donner la hiérarchies des classes `Preparation`, `Ingredient`, `Operation`, `AjoutMelange`, `CuissonAuFour`, `Recette` et `PreparationComposite`. Attention, la classe `PreparationComposite` doit être à la racine de tous les objets manipulant des ensembles de préparations. [La réponse à cette question peut être donnée en même temps que celle de la question suivante].

```
Preparation <<ABS>>
  - nom : String
  + Preparation(String)
  + String getNom()

Ingredient extends Preparation
    -  double quantite;
    -  private String unite;
    + Ingredient(String nom, double quantite)
    + Ingredient(String nom)
    + double getQuantite()
    + String getUnite()

PreparationComposite extends Preparation
    - Preparation[] tab;
    + PreparationComposite(Preparation[] tab)
    + PreparationComposite(Preparation[] tab, String nom)
    + Preparation[] getTableau()

      Recette extends PreparationComposite
          + Recette(Preparation[] tab, String nom)
          // pas de gestion du nom ici !!

      Operation extends PreparationComposite <<ABS>>
          + Operation(Preparation[] tab)

            AjoutMelange extends Operation
               + AjoutMelange(Preparation[] tab)

            CuissonAuFour extends Operation
               + CuissonAuFour(Preparation[] tab, String commentaires, double temperature)
```

Q 5.2 (3.5 pts) Réfléchir aux attributs, aux constructeurs et méthodes présents dans chaque classe, ainsi qu'aux classes abstraites et concrètes. Donner les signatures des méthodes et les déclarations d'attributs. [Pas de code]

Q 5.3 (2 pts) Donner le code de la méthode standard `equals` de la classe `Ingredient`, qui vérifie si les noms, les unités et les quantités sont les mêmes pour 2 ingrédients. ATTENTION, si la quantité d'un des deux ingrédients est réglée sur -1, seul le nom sera pris en compte. [Pour simplifier, nous considérons que les String ne sont jamais `null` dans cette classe.]

On met très peu de point dès que la structure est mauvaise.
Quand la structure est bonne, on regarde la gestion des -1 et des .equals entre string.

```
1     public boolean equals(Object obj) { // -2 si mauvaise signature
2         if (this == obj) //OPT
3             return true;
```

```
4         if (obj == null)
5             return false;
6         if (getClass() != obj.getClass()) // -1 si oubli
7             return false;
8         Ingredient other = (Ingredient) obj; // -1.5 si oubli
9         if (quantite != other.quantite && quantite != -1 && other.quantite != -1)
10            return false; // -1 si mauvaise gestion des contraintes
11        if(!nom.equals(other.nom))
12            return false;
13        if(!unite.equals(other.unite) && quantite != -1 && other.quantite != -1)
14            return false;
15        return true;
16    }
```

Q 5.4 (1.5 pt) Nous souhaitons créer une classe `Recette` qui a un nom et permet de gérer un ensemble de `Preparation`. De quelle classe doit-elle hériter ? Quel(s) attribut(s) doit-elle avoir ?

0.5 pt : Elle doit hériter de `PreparationComposite`
1 pt : Aucun attribut n'est nécessaire : le nom est celui de la `Preparation`

Q 5.5 (5 pts) Dans la classe `PreparationComposite`, nous souhaitons récupérer tous les ingrédients de base (de la classe `Ingredient`). Donner le code de la méthode `ArrayList<Ingredient> getAllIngredients()` qui effectue cette opération.
Note : attention à ne renvoyer que des `Ingredient`.
Note 2 : une solution consiste à procéder récursivement en exploitant la méthode `getTableau()`, pour cela vous devrez définir une seconde méthode, elle-même invoquée par `getAllIngredients()`.

Un peu dur... Histoire de donner du boulot aux très bons ! Le sujet sera noté sur plus de 60 au final...

```
1     public ArrayList<Ingredient> getAllIng(){
2         ArrayList<Ingredient> tab = new ArrayList<Ingredient>();
3         addIngredients(tab, getTableau());
4         return tab;
5     }
6
7     private void addIngredients(ArrayList<Ingredient> tab , Preparation[] toAdd){
8         for(Preparation p:toAdd){
9             if(p instanceof Ingredient)
10                tab.add((Ingredient) p);
11            else if (p instanceof PreparationComposite)
12                addIngredients(tab, ((PreparationComposite) p).getTableau());
13            else
14                System.out.println("ATTENTION");
15        }
```

Q 5.6 (4 pts) [Faire l'hypothèse que vous disposez de la méthode précédente]. Nous voulons modéliser une cuisine de nouvelle génération, gérant l'ensemble des `Preparation` disponibles en ce moment et proposant automatiquement des `Recette`.
Proposer une classe `CuisineIntelligente` disposant d'une méthode `double evaluationRecette(Recette r)` donnant le pourcentage d'ingrédients disponibles pour cette recette. Réfléchir à l'intégration dans la hiérarchie de classes existante et donner un constructeur cohérent.
Note : Pour plus de simplicité, considérer que les quantités des ingrédients dans la `CuisineIntelligente` sont toutes à -1. Vous pouvez ainsi utiliser directement la méthode `contains` de `ArrayList` qui est documentée en fin de page précédente.

```
1 public class Cuisine extends PreparationComposite { // 1 pt pour le choix de PreparationComposite
2
3     public Cuisine(Preparation[] tab) { // 1 pt  constructeur
4         super(tab);
5     }
6
7     public double evaluationRecette(Recette r){ // 2 pts
8         int cpt = 0;
9         ArrayList<Ingredient> allIng = this.getAllIng();
```

```
10          for (Ingredient i : r.getAllIng())
11              if(allIng.contains(i))
12                  cpt++;
13          return (double) cpt/ r.getAllIng().size();
14      }
```