Examen

Nom : **Prénom :**

Préambule : Ne pas mettre de commentaires de documentation sauf s'ils sont nécessaires à la compréhension.

Les exercices sont relativement indépendants.

Barème indicatif :

Exercice	1	2	3	4	5	6	7
Points	4	1,5	1	4	4	4	1,5

1 Compréhension du cours

Exercice 1 Répondre de manière concise aux questions suivantes.

1.1 Considérons la déclaration du listing suivant. Nous supposons qu'elle apparaît dans la classe `Point` vue en cours, TD et TP.

```
1 public class Point {
2    public static final Point origine = new Point(0, 0);
3    ...
4 }
```

1.1.1 Expliquer ce que signifient les mots-clés **`public`**, **`static`** et **`final`**.

1.1.2 Indiquer, en justifiant la réponse, si on peut être sûr que l'attribut `origine` de la classe `Point` aura toujours pour coordonnées $(0, 0)$.

1.2 Intéressons nous à JUnit dans sa version 3.8.1 (la version utilisée en TP).

1.2.1 Rappeler en 4 lignes maximum l'objectif et l'intérêt de JUnit.

1.2.2 Écrire une classe de test s'appuyant sur JUnit 3.8.1 et définissant deux méthodes de test qui vérifient respectivement que :

- `Integer.parseInt("10")` retourne la valeur 10,
- `Integer.parseInt("10x")` lève l'exception `NumberFormatException`. Ce test doit donc échouer si aucune exception n'est levée ou si c'est une autre exception qui est levée.

2 Partage de liste en Java et utilisation du patron Proxy

L'objectif de ces exercices est de s'intéresser à différentes stratégies pour transmettre un objet en étant sûr que les modifications éventuellement faites sur l'objet par la méthode appelée ne seront pas visibles de la méthode appelante. On veut donc obtenir l'équivalent d'un passage

par valeur sur l'objet. L'objet transmis sera une liste identique à celles vues en cours et dont le diagramme de classe est rappelé à la figure 1. Le code de `Liste` et `ListeTab` est donné aux listings 1 et 2.

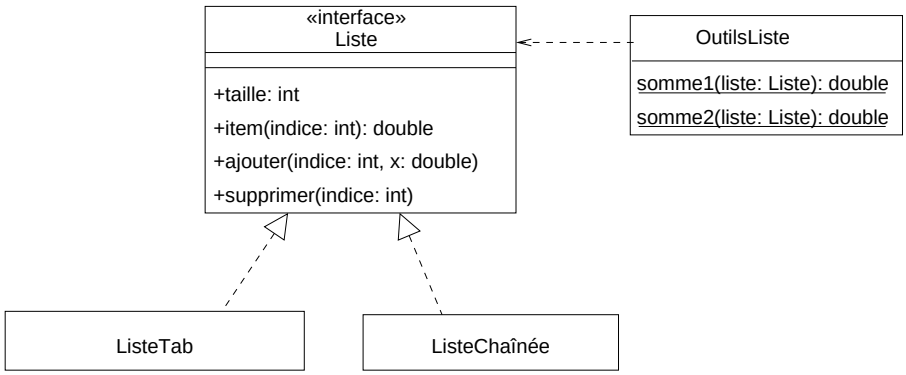

FIGURE 1 – Diagramme de classe des listes

Listing 1 – La classe `Liste`

```
1  /** Spécification d'une liste. @version 1.5 */
2  public interface Liste {
3     /** Obtenir la taille de la liste.
4      * @return nombre d'éléments dans la liste */
5     int taille();
6
7     /** Obtenir un élément de la liste.
8      * @param indice position de l'élément */
9     double item(int indice);
10
11    /** Ajouter un élément dans la liste.
12     * @param indice indice où doit se trouver le nouvel élément
13     * @param x élément à insérer */
14    void ajouter(int indice, double x);
15
16    /** Supprimer un élément de la liste.
17     * @param indice indice de l'élément à supprimer */
18    void supprimer(int indice);
19 }
```

Listing 2 – La classe `ListeTab`

```
1  /** Liste simple utilisant un tableau pour stocker les éléments. */
2  public class ListeTab implements Liste {
3     private double[] elements; // les éléments de la liste
4     private int nb;      // la taille de la liste
5
```

```
6     /** Construire une liste vide.
7      * @param capacite capacité initiale de la liste
8      */
9     public ListeTab(int capacite) {
10       this.elements = new double[capacite];
11       this.nb = 0;      // la liste est initialement vide
12    }
13
14    public int taille() {
15       return this.nb;
16    }
17
18    public double item(int index) {
19       return this.elements[index];
20    }
21
22    public void supprimer(int index) {
23       System.arraycopy(this.elements, index+1,
24             this.elements, index, this.nb-index-1);
25       this.nb--;
26    }
27
28    public void ajouter(int index, double x) {
29       if (this.nb >= this.elements.length) { // tableau trop petit !
30          // agrandir le tableau : pourrait être plus efficace !
31          double[] nouveau = new double[this.nb+3]; // 3 arbitraire
32          System.arraycopy(this.elements, 0, nouveau, 0, this.nb);
33          this.elements = nouveau;
34       }
35       // décaler les éléments à partir de index
36       System.arraycopy(this.elements, index,
37             this.elements, index+1, this.nb-index);
38       // ranger le nouvel élément
39       this.elements[index] = x;
40       this.nb++;
41    }
42
43    public String toString() {
44       // ... code non donné ...
45    }
46
47 }
```

Exercice 2 : Comprendre le problème posé

Pour bien comprendre le problème posé, nous nous intéressons à la classe `OutilsListe` (listing 3) qui définit deux méthodes `somme1` et `somme2` pour calculer la somme des éléments d'une liste. La classe `ClientOutilsListe` (listing 4) utilise la classe `OutilsListe`.

2.1 Donner le résultat de l'exécution de la classe `ClientOutilsListe`.

2.2 Expliquer pourquoi les deux appels à la méthode `somme1` donnent des résultats différents. Les exercices qui suivent auront pour objectif d'étudier des solutions pour que les deux appels à `somme1` donnent le même résultat.

2.3 Pour écrire `somme1`, on parcourt les éléments de la liste. Indiquer le patron de conception (design pattern) qui est préconisé quand on souhaite parcourir tous les éléments d'une structure de données.

Listing 3 – La classe `OutilsListe`

```
1  public class OutilsListe {
2
3     public static double somme1(Liste l) {
4        double resultat = 0;
5        for (int i = 0; i < l.taille(); i++) {
6           resultat += l.item(i);
7        }
8        return resultat;
9     }
10
11    public static double somme2(Liste l) {
12       double resultat = 0;
13       while (l.taille() > 0) {
14          resultat += l.item(0);
15          l.supprimer(0);
16       }
17       return resultat;
18    }
19
20 }
```

Listing 4 – La classe `ClientOutilsListe`

```
1  public class ClientOutilsListe {
2
3     public static void main(String[] args) {
4        Liste l = new ListeTab(5);
5        l.ajouter(0, 3.5);
6        l.ajouter(1, -1.0);
7
8        System.out.println("somme1 = " + OutilsListe.somme1(l));
9        System.out.println("somme2 = " + OutilsListe.somme2(l));
10       System.out.println("somme1 = " + OutilsListe.somme1(l));
11    }
12
13 }
```

Exercice 3 : Copier les listes

Une première stratégie consiste à copier la liste avant de la transmettre. Nous ajoutons une méthode `copie` [1] sur l'interface `Liste`. Elle consiste à construire une copie de la liste, c'est-à-dire une nouvelle liste avec les mêmes éléments aux mêmes positions.

3.1 Écrire la méthode `copie` de la classe `ListeTab`.

3.2 Indiquer comment modifier le listing 4 pour que la liste `l` ne soit pas modifiée à la fin de l'exécution du programme.

Exercice 4 : Comprendre le patron de conception Proxy

Le patron de conception Proxy [2] est utilisé quand on veut ajouter un intermédiaire, un *proxy*, entre le client et l'objet qu'il souhaite manipuler. Le diagramme de classe de ce patron est donné à la figure 2. Le *client* souhaite appeler une *requête* sur un objet, le *sujet réel*, qui contient l'implantation de la *requête*. Cependant, au lieu de s'adresser directement au *sujet réel*, le *client* interroge le mandataire (le proxy) qui transmettra la *requête* au *sujet réel*. Le proxy est généralement chargé

1. En Java, on utiliserait plutôt la méthode `clone` de la classe `Object`. Ici, nous nous interdisons d'utiliser `clone`.
2. Le patron Proxy est aussi appelé Procuration ou Mandataire.

de prendre en compte des aspects non fonctionnels de l'application comme par exemple réaliser des traces, gérer un appel à distance, calculer des statistiques, se comporter comme un cache, etc. L'intérêt du patron Proxy est que le client ne sait pas qu'il accède au sujet réel au moyen d'un proxy. Les aspects non fonctionnels sont donc ajoutés de manière transparente pour le client.

Pour bien comprendre le patron Proxy, nous allons l'utiliser pour vérifier que les opérations de la liste sont appelées avec des paramètres cohérents. Nous allons ainsi définir un proxy CheckedListe.

4.1 Dessiner le diagramme de classe dans lequel doivent apparaître l'interface `Liste` (le sujet), les classes `ListeTab` et `ListeChainee` (le sujet réel), `CheckedListe` (le proxy) et une classe de test (le client).

4.2 Dessiner le diagramme de séquence qui correspond au scénario suivant.

1. le programme de test crée un objet `ListeTab` appelé `lt`,
2. le programme de test crée un objet `CheckedListe` appelé `cl` pour la liste `lt`,
3. le programme de test ajoute l'élément 3.14 à la position 0 en s'adressant au proxy,
4. le proxy vérifie que la position 0 est bien comprise entre 0 et la taille de la liste `lt`,
5. le proxy appelle l'opération `ajouter` sur l'objet `lt`,
6. le proxy vérifie que l'élément à la position 0 de la liste `lt` vaut 3.14,
7. le programme de test demande la taille de la liste en s'adressant au proxy,
8. le proxy transfère la demande à la liste `lt`,
9. le proxy retourne la valeur reçue au programme de test,

4.3 Écrire la classe `CheckedListe`. On ne donnera pas le code des méthodes `supprimer`, `item` et `copie`. On utilisera l'instruction `assert exprBooléenne;` qui vérifie que `exprBooléenne` vaut vrai. Par exemple, `assert x != 0;` vérifie que `x` est non nul.

4.4 Indiquer deux techniques vues en cours pour exprimer des conditions sur les paramètres d'entrée (ou de sortie) d'une méthode.

Exercice 5 : Interdire toute modification de la liste

Faire une copie de la liste est généralement coûteux. Aussi, quand une méthode n'est pas sensée

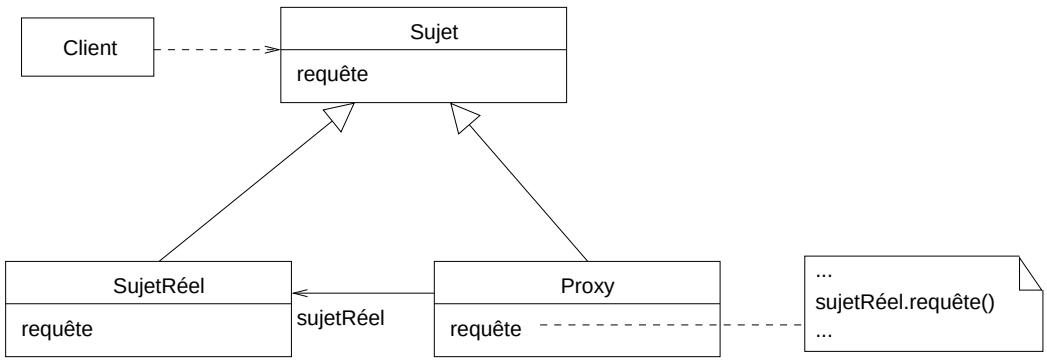

FIGURE 2 – Le diagramme de classe du patron Proxy

modifier la liste on peut souhaiter ne pas faire la copie inutile. C'est par exemple le cas de la méthode `somme1` (figure 1 et listing 3). Malheureusement, il n'est pas possible en Java de spécifier qu'une méthode n'a pas le droit de modifier la liste passée en paramètre [3]. Ainsi, le programmeur n'est pas à l'abri d'une méthode qui modifiera quand même la liste (`somme2` par exemple). Dans cet exercice, nous envisageons des techniques pour s'assurer que la méthode ne pourra effectivement pas modifier la liste.

5.1 *Utilisation d'une interface `ListeNonModifiable`.* La première idée consiste à définir une interface `ListeNonModifiable`. Cette interface ne spécifie que les opérations qui ne modifient pas le contenu de la liste (donc `taille`, `item` et `copie`). L'interface `Liste` hérite de `ListeNonModifiable`. La méthode `somme1` prend alors en paramètre un objet de type `ListeNonModifiable`.

5.1.1 Modifier le diagramme de classe de la figure 1 pour prendre en compte cette solution. Répondre directement sur le sujet.

5.1.2 Cette solution ne fonctionne pas. Le montrer en écrivant une méthode de classe `vider` qui prend en paramètre un objet `ListeNonModifiable` et en supprime tous les éléments.

5.2 *Utilisation d'un proxy `ReadOnlyListe`.* La deuxième idée consiste à s'appuyer sur le patron Proxy pour interdire l'utilisation des opérations `ajouter` et `supprimer`. Si ces opérations sont appelées, une exception `OperationInterditeException` (`OIP` en abrégé) signalera que l'accès à ces opérations est interdit.

5.2.1 Indiquer, en justifiant la réponse, si l'exception `OperationInterditeException` peut être placée sous le contrôle du compilateur.

5.2.2 Écrire la classe `OperationInterditeException`.

5.2.3 Expliquer pourquoi la méthode `copie` de `ReadOnlyListe` peut se contenter de renvoyer **`this`** sans créer un nouvel objet `ReadOnlyListe`.

5.2.4 Écrire la classe `ReadOnlyListe`.

Exercice 6 : Automatiser la copie de la liste

Les deux stratégies précédentes (copie systématique dans l'exercice 3 et interdiction de modifier la liste dans l'exercice 5) fonctionnent mais c'est au programmeur de savoir quelle version de la liste fournir (soit une copie de la liste, soit une instance de `ReadOnlyListe`).

Une autre stratégie consiste à faire systématiquement une copie, mais la copie d'un proxy qui s'occupera de copier la liste si c'est nécessaire. Copier la liste est nécessaire si une opération de modification est appelée et si la liste est partagée par plusieurs proxy. On parle alors de *copy on write (cow)*. On appellera `CopyOnWriteListe` ce nouveau proxy. Le listing 5 est la version modifiée du listing 6 utilisant ce proxy. Pour `somme1`, la liste instance de `ListeTab` ne sera pas copiée (pas d'opération de modification appelée) alors que pour `somme2` une copie sera réalisée.

Le proxy `CopyOnWriteListe` est défini par :

1. la liste dont il contrôle l'accès (nous l'appellerons la liste réelle) et
2. un compteur du nombre d'utilisateur de cette liste (compteur de références).

3. Ceci existe dans certains langages. C'est par exemple le cas de Ada (et UML) avec le mode de passage de paramètre **`in`** ou de C++ avec la notion de **`const`**.

Listing 5 – La classe `ClientOutilsListe` avec utilisation de `CopyOnWriteListe`

```
1  public class ClientOutilsListe {
2
3     public static void main(String[] args) {
4        Liste l = new CopyOnWriteListe(new ListeTab(5));
5        l.ajouter(0, 3.5);
6        l.ajouter(1, -1.0);
7
8        System.out.println("somme1 = " + OutilsListe.somme1(l.copie()));
9        System.out.println("somme2 = " + OutilsListe.somme2(l.copie()));
10       System.out.println("somme1 = " + OutilsListe.somme1(l.copie()));
11    }
12
13 }
```

Si une copie du proxy est demandée, il suffit de :

1. créer un nouvel objet `CopyOnWriteListe` qui référence la même liste réelle et le même compteur que le proxy à copier et
2. incrémenter le compteur de références puisqu'il y a un nouvel utilisateur de la liste.

La liste réelle et le compteur de références sont alors partagés par plusieurs proxy.

Lorsqu'une opération de modification est exécutée, il faut s'assurer que la modification ne sera pas visible des autres utilisateurs. Ainsi, s'il y a plusieurs références sur la liste, il faut faire une copie de la liste réelle propre à ce proxy et indiquer que le nombre de références sur cette copie est de 1. Le nombre de références sur la liste réelle d'origine est diminué de 1.

Le listing 6 donne un exemple d'utilisation de ce proxy et le listing 7 le résultat de son exécution. Pour l'affichage d'une liste de type `CopyOnWriteListe`, on affiche la valeur du compteur de références entre parenthèses, puis le contenu de la liste et enfin, le type et l'adresse de la liste.

6.1 Pour définir le compteur de référence, on ne peut pas utiliser une variable de type **`int`** car elle doit pouvoir être partagée par plusieurs instances de `CopyOnWriteListe`. Aussi, nous définissons une classe `MutableInteger` qui représentera ce compteur. Elle définit les opérations `get`, `set` et `add` pour respectivement obtenir la valeur de l'entier `MutableInteger`, la modifier et lui ajouter un entier (**`int`**).

Écrire la classe `MutableInteger`.

6.2 Dessiner le diagramme de classe qui fait apparaître les classes `MutableInteger`, `Liste` et `CopyOnWriteListe`. On ne fera apparaître ni attribut, ni constructeur, ni opération.

6.3 Écrire la classe `CopyOnWriteListe`. Elle devra définir une méthode `assurerListeNonPartagee` qui assure que la liste réelle n'est pas partagée. Cette méthode réalise donc la copie de la liste et la mise à jour des compteurs de références s'il y a lieu.

6.4 Expliquer à quoi correspond la méthode `finalize` de la classe `Object` et indiquer s'il est utile de la définir pour `CopyOnWriteListe`.

Exercice 7 : Généricité

Modifier l'interface `Liste` et les classes `ListeTab` et `ClientOutilsListe` des listings 1, 2 et 4 pour qu'elles soient génériques, paramétrées par le type des éléments qu'elles contiennent. Répondre directement sur le sujet.

Listing 6 – La classe `ExempleCopyOnWriteListe`

```
1  public class ExempleCopyOnWriteListe {
2
3     private static Liste cow1, cow2, cow3;
4
5     public static void afficher() {
6        System.out.println("cow1 = " + cow1);
7        System.out.println("cow2 = " + cow2);
8        System.out.println("cow3 = " + cow3);
9        System.out.println();
10    }
11
12    public static void main(String[] args) {
13       ListeTab lt = new ListeTab(10);
14       lt.ajouter(0, 1);
15       lt.ajouter(1, 2);
16       System.out.println("lt = " + lt);
17                                              afficher();
18       cow1 = new CopyOnWriteListe(lt);       afficher();
19       cow2 = cow1.copie();                   afficher();
20       cow3 = cow1.copie();                   afficher();
21       cow1.ajouter(0, 5);                    afficher();
22       Liste cow4 = cow2.copie();             afficher();
23       cow2.supprimer(0);                     afficher();
24    }
25 }
```

Listing 7 – Résultat de l'exécution de la classe `ExempleCopyOnWriteListe`

```
1  lt = [ 1.0, 2.0 ] @ ListeTab@e83912
2  cow1 = null
3  cow2 = null
4  cow3 = null
5
6  cow1 = (1)->[ 1.0, 2.0 ] @ ListeTab@e83912
7  cow2 = null
8  cow3 = null
9
10 cow1 = (2)->[ 1.0, 2.0 ] @ ListeTab@e83912
11 cow2 = (2)->[ 1.0, 2.0 ] @ ListeTab@e83912
12 cow3 = null
13
14 cow1 = (3)->[ 1.0, 2.0 ] @ ListeTab@e83912
15 cow2 = (3)->[ 1.0, 2.0 ] @ ListeTab@e83912
16 cow3 = (3)->[ 1.0, 2.0 ] @ ListeTab@e83912
17
18 cow1 = (1)->[ 5.0, 1.0, 2.0 ] @ ListeTab@fd13b5
19 cow2 = (2)->[ 1.0, 2.0 ] @ ListeTab@e83912
20 cow3 = (2)->[ 1.0, 2.0 ] @ ListeTab@e83912
21
22 cow1 = (1)->[ 5.0, 1.0, 2.0 ] @ ListeTab@fd13b5
23 cow2 = (3)->[ 1.0, 2.0 ] @ ListeTab@e83912
24 cow3 = (3)->[ 1.0, 2.0 ] @ ListeTab@e83912
25
26 cow1 = (1)->[ 5.0, 1.0, 2.0 ] @ ListeTab@fd13b5
27 cow2 = (1)->[ 2.0 ] @ ListeTab@118f375
28 cow3 = (2)->[ 1.0, 2.0 ] @ ListeTab@e83912
```