TP de Compléments en Programmation Orientée Objet n° 1 : Objets, classes et encapsulation
(Correction)

Exercice 1 : Généralités

Faites le QCM qui est sur Moodle. La note obtenue est purement indicative, elle ne sera pas prise en compte dans la note finale. Remarquez qu'il est noté de manière plus gentille que les QCM notés qui seront faits en cours : ici une réponse juste compte 1, une fausse compte -0,5 et pas de réponse 0. (En cours les mauvaises réponses comptent pour -1)

Exercice 2 : statique et non-statique

Mettez "V" (pour vrai) ou "F" pour faux dans la case.

```
1   class Truc {
2       static int v1 = 0;
3       int v2 = 0;
4       public int getV1() { return v1; }
5       public int getV2() { return v2; }
6       public Truc() {
7           v1++; v2++;
8       }
9   }
10
```

```
11  public class Main {
12      public static void main(String args[]) {
13          System.out.println(new
                Truc().getV1());
14          System.out.println(new
                Truc().getV2());
15          System.out.println(new
                Truc().getV1());
16      }
17  }
```

F La ligne 14 affichera "2". F La ligne 15 affichera "1".

Correction : `v1` et `v2` n'ont pas le même statut. `v1` est statique et donc n'existe qu'en un seul exemplaire, incrémenté à chaque instanciation de `Truc` (donc 3 fois avant la ligne 15) . Donc La ligne 15 affiche "3".
`v2`, elle, est un attribut d'instance, donc un nouvel exemplaire existe pour chaque nouvelle instance de `Truc`, dont la valeur vaut 1 à la sortie du constructeur. Donc la ligne 14 affiche "1".

Exercice 3 : Constructeurs des classes imbriquées

Soient les classes :

```
1   public class A{
2       private int a;
3
4       public class AA{
5           private int a;
6           public AA(int y){ this.a = y;}
7       }
8
9       public static class AB{
10          private int b;
11          public AB(int x){ this.b = x;}
12      }
13
```

```
14      public A(int a){
15      this.a = a;
16      }
17  }
18  public class Main{
19      public static void main(String[] args){
20          A unA = new A(2);
21          // A.AA unAA = new A.AA(3);
22          // A.AB unAB = new A.AB(4);
23          // A.AA autreAA = unA.new AA(3);
24          // A.AB autreAB = unA.new AB(4);
25      }
26  }
```

— Parmi les lignes 21 à 24, quelles sont celles que le programme compile encore quand on les « dé-commente » ?

Correction : Les lignes 22 et 23 compilent.

Exercice 4 : Encapsulation et sûreté

Voici deux classes avec leur spécification. Pour chaque cas :
— soit la spécification est satisfaite par la classe, dans ce cas, justifiez-le ;
— soit la spécification n'est pas satisfaite, dans ce cas écrivez un programme qui la met en défaut (sans modifier la classe fournie), puis proposez une rectification de la classe.

1.
```
1 public class EvenNumbersGenerator {
2     static int MAX = 42;
3     public int previous = 0;
4     public int next() {
5         previous += 2; previous %= MAX;
6         return previous;
7     }
8 }
```

Spécification : la méthode `next` ne retourne que des entiers pairs.

2.
```
1  public class VectAdditioner {
2      private Point sum = new Point();
3
4      public void add(Point p) {
5          sum.x += p.x; sum.y += p.y;
6      }
7
8      public Point getSum() {
9          return sum;
10     }
11 }
```

Spécification : la méthode `getSum` retourne la somme de tous les vecteurs qui ont été passés en paramètre par la méthode `add` depuis l'instanciation.

Correction :

1. `EvenNumbersGenerator` fait ce qu'elle promet tant que l'attribut `previous` n'est pas modifié depuis l'extérieur.
   Par exemple :
   ```
   1 EvenNumbersGenerator eng = new EvenNumbersGenerator();
   2 eng.previous = 1;
   3 System.out.println(eng.next()); // affiche 3
   ```
   Une attaque similaire est possible en modifiant `MAX`.
   La solution : ajouter **`private`** et/ou **`final`** à la ligne 2 et remplacer **`public`** par **`private`** à la ligne 3 empêche cela.
   Après cette modification, on a la garantie que `MAX` ne sera plus modifiée et que `previous` ne sera modifiée que par la méthode `next`. Or cette dernière semble bien programmée, vu qu'elle ne fait qu'appliquer les opérations + 2 et modulo 42 à l'attribut `previous` (opérations qui préservent la parité).
   (Remarques en avance sur le cours :) Cela dit, si cette méthode est exécutée 2 fois en même temps (sur plusieurs *threads*), les différentes opérations peuvent s'entrelacer. En l'occurence, ici, cela ne casserait pas la garantie de parité. Par ailleurs, des accès en compétition (i.e. : lecture et écriture d'une même variable depuis des *threads* différents, sans synchronisation) auraient lieu, ce qui peut avoir de mauvaises conséquences (cf. cours sur la concurrence, à venir).
   Ajouter **`volatile`** devant la déclaration de `previous` règle ce dernier problème, sans

pour autant empêcher les entrelacements bizarres. Pour garantir des propriétés plus fortes que la parité, il faudrait recourir à d'autres primitives de synchronisation, comme `synchronized`, afin d'obtenir d'atomicité de ce calcul.

2. Ici, le problème, c'est que malgré le `private`, on peut malgré tout obtenir, par appel à `getSum()` une copie de la référence contenue dans l'attribut `sum`.
   Si on fait :

```
1       VectAdditioner va = new VectAdditioner();
2       va.getSum().x = -12;
3       System.out.prinln(va.getSum());
```

   Alors s'affichera `(-12, 0)`, alors même qu'on n'a pas encore appelé `add`.
   La solution : ne pas partager de référence. Pour cela, modifier la méthode `getSum` afin qu'elle retourne une référence vers un nouvel objet au lieu d'une simple copie de `sum` :

```
1   public Point getSum() {
2      return new Point(sum.x, sum.y);
3   }
```

   (Remarques en avance sur le cours :) Là encore, il reste des problèmes liés à la concurrence. Si deux appels à `add` s'entrelaçaient, de l'information pourrait être perdue. Pour corriger, il suffit de rendre `add` atomique en ajoutant `synchronized` à sa déclaration. Il faudrait aussi ajouter `synchronized` à `getSum` pour s'assurer que le dernier appel à `add` est terminé quand on copie `sum`.

Toute les considérations en rapport à la programmation concurrente seront discutées plus tard dans le cours. Le problème n'est mentionné dans ce corrigé que dans un souci d'exactitude.

Exercice 5 : Nombres complexes

A commencer dans ce TP et à finir dans le prochain TP.
Pour le vocabulaire, référez-vous à `https://fr.wikipedia.org/wiki/Nombre_complexe`.

1. Écrivez une classe `Complexe`, avec :
   — les attributs (`double`) : parties réelle et imaginaire du nombre (`static` ou pas ?) ;
   — le constructeur, prenant comme paramètres les parties réelle et imaginaire du nombre ;
   — la méthode `public String toString()`, permettant de convertir un complexe en chaîne de caractères lisible par l'humain ;
   — les opérations arithmétiques usuelles (somme, soustraction, multiplication, division) ;
   — le test d'égalité (méthode `public boolean equals(Object other)`) ;
   — les fonctions et accesseurs spécifiques aux complexes : partie réelle, partie imaginaire, conjugaison, module, argument...
   Vous pouvez vous aider des fonctionnalités de génération de code de votre IDE.
   Attention, style demandé : attributs non modifiables (si vous savez le faire, faites une vraie classe immuable), les opérations retournent de nouveaux objets.
2. Ajoutez à votre classe :
   — des attributs (constants : vous pouvez ajouter `final`) pour les valeurs les plus courantes du type `Complexe` : à savoir 0, 1 et i (le nombre i tel que $i^2 = -1$).
   Ces attributs doivent-ils être `static` ou non ?

— une méthode (fabrique statique)

```
1  public static Complexe fromPolarCoordinates(double rho, double theta)
```

qui construit un complexe depuis son module ρ et son argument θ (on rappelle que la partie réelle vaut alors $\rho\cos\theta$ et la partie imaginaire $\rho\sin\theta$).
Remarquez que cette méthode joue le rôle d'un constructeur. Pourquoi ne pas avoir fait plutôt un autre constructeur alors ? (essayez de compiler avec 2 constructeurs puis expliquez pourquoi ça ne marche pas)

Correction :

```
1
2   public final class Complexe {
3
4       /*
5       * on pourrait presque mettre les attributs en public sans dégâts dans ce
6       * cas (final), mais faire ainsi nous empêcherait dans des évolutions
7       * futures de changer la représentation interne, donc nous allons quand-même
8       * définir les getteurs.
9       */
10      public final double re, im;
11      public final static Complexe I = new Complexe(0, 1);
12
13      public double getRe() {
14          return re;
15      }
16
17      public double getIm() {
18          return im;
19      }
20
21      /*
22      * Constructeur privé pour forcer à utiliser les méthodes fabrique statiques
23      */
24      private Complexe(double re, double im) {
25          this.re = re;
26          this.im = im;
27      }
28
29      @Override
30      public String toString() {
31          return "(" + re + " + " + im + "i)";
32      }
33
34      /*
35      * NB: il faudrait normalement redéfinir en même temps la méthode
36      * hashCode() de façon cohérente (EJ 3 Item 11).
37      */
38      @Override
39      public boolean equals(Object autre) {
40          if (autre == null || !(autre instanceof Complexe)) return false;
41          Complexe autreComplexe = (Complexe) autre;
42          return re == autreComplexe.re
43              && im == autreComplexe.im;
44      }
45
46      public Complexe plus(Complexe autre) {
47          return new Complexe(re + autre.re, im + autre.im);
48      }
49
50      public Complexe moins(Complexe autre) {
51          return new Complexe(re - autre.re, im - autre.im);
52      }
53
54      public Complexe fois(Complexe autre) {
```

```
55          return new Complexe(re * autre.re - im * autre.im, re * autre.im + im *
                autre.re);
56      }
57
58      public Complexe divisePar(Complexe autre) {
59          if (autre.egale(ZERO))
60              throw new ArithmeticException("Division by zero.");
61          double d = autre.re * autre.re + autre.im * autre.im;
62          double r = re * autre.re;
63          double i = +im * autre.im;
64          return new Complexe((r + i) / d, (r - i) / d);
65      }
66
67      public Complexe conjugue() {
68          return new Complexe(re, -im);
69      }
70
71      public double module() {
72          return Math.sqrt(re * re + im * im);
73      }
74
75      public double argument() {
76          if (re == 0) {
77              if (im == 0) return Double.NaN;
78              else if (im < 0) return -Math.PI / 2;
79              else return Math.PI / 2;
80          }
81          else if (re > 0) return Math.atan(im / re);
82          else if (im >= 0) return Math.atan(im / re) + Math.PI;
83          else return Math.atan(im / re) - Math.PI;
84      }
85
86      private static final Complexe ZERO = new Complexe(0, 0);
87
88      private static final Complexe UN = new Complexe(1, 0);
89
90      private static final Complexe I = new Complexe(0, 1);
91
92      public static Complexe fromPolarCoordinates(double rho, double theta) {
93          return new Complexe(rho * Math.cos(theta), rho = Math.sin(theta));
94      }
95
96      /*
97      * Ne fait rien de plus que le constructeur, mais c'est plus équilibré de
98      * mettre les deux modes de contsruction sur un pied d'égalité dans l'API.
99      */
100     public static Complexe fromRealAndImaginary(double re, double im) {
101         return new Complexe(re, im);
102     }
103 }
```

3. Améliorez l'encapsulation de votre classe, afin de permettre des évolutions ultérieures sans « casser » les clients/dépendents de celle-ci : en l'occurrence, les attributs doivent être privés et des accesseurs publics [1] doivent être ajoutés pour que la classe reste utilisable.
4. Testez en écrivant un programme (méthode `main()`, dans une autre classe), qui fait entrer à l'utilisateur une séquence de nombre complexes et calcule leur somme et leur produit. Améliorez le programme pour permettre la saisie des nombres au choix, via leurs parties

1. Remarquez qu'il n'y a pas de raison de favoriser le couple parties réelle/imaginaire par rapport au couple module/argument (les deux définissent de façon unique un nombre complexe) ; et qu'il faut donc considérer ce dernier couple comme un couple de propriétés, pour lequel il faudrait utiliser aussi la notation `get`. Ainsi, cette classe aurait 4 propriétés (peu importe si elles sont redondantes : les attributs ne le sont pas ; cela limite les risques d'incohérences).

réelles et imaginaires ou via leurs coordonnées polaires.

5. Écrivez une version « mutable » de cette classe (il faut donc des méthodes `set` pour chacune des propriétés).
   Changez la signature [2] et le comportement des méthodes des opérations arithmétiques afin que le résultat soit enregistré dans l'objet courant (`this`), plutôt que retourné.

2. Si la méthode déjà programmée est `static` rendre non statique et enlever un paramètre ; dans tous les cas retourner `void`.