Université Paris Cité

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

Exercice 1 :

Les affirmations suivantes sont-elles rigoureusement exactes ?

1. Un même code-octet JVM est exécutable sur plusieurs plateformes physiques (x86, PPC, ARM, ...).
2. La JVM interprète du code source Java.

   Correction : La JVM interprète seulement (ou bien compile à la volée, cf. JIT) du code-octet, mais en aucun cas du code source. C'est le rôle du compilateur (javac) de comprendre le code source.

3. Le mot-clé `this` est une expression qui s'évalue comme l'objet sur lequel la méthode courante a été appelée.
4. Toute classe dispose d'un constructeur par défaut, sans paramètre.

   Correction : Toute classe non munie d'un constructeur écrit par le programmeur se verra ajouter automatiquement le constructeur par défaut par le compilateur.
   Les autres classes n'ont pas ce constructeur par défaut, mais seulement ceux que le programmeur aura écrits.

5. On écrit `public` devant la déclaration d'une variable locale pour qu'elle soit visible depuis les autres classes.

   Correction : Les variables locales sont... locales ! Ceci signifie qu'elles n'ont du sens que dans le bloc où elles sont déclarées. Les modificateurs de visibilité ne s'y appliquent donc pas.

6. Dès lors qu'un objet n'est plus utilisé, il faut penser à demander à Java de libérer la mémoire qu'il occupe.

   Correction : Non, le ramasse-miettes détermine automatiquement quels objets ne sont plus référencés et peuvent donc être libérés (ce qu'il va donc faire périodiquement sans qu'on ait à le demander).

7. La durée de vie d'un attribut statique est liée à celle d'une instance donnée de la classe où il est défini.

   Correction : Non, la durée de vie d'un attribut statique est liée à la durée d'existence de la classe dans la mémoire de la JVM (typiquement : toute la durée de l'exécution du programme).

Exercice 2 : statique et non-statique

Qu'est-ce que affichent les lignes 13, 14 et 15 dans le code suivant ?

```
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 }
18
```

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.
```
public class EvenNumbersGenerator {
    static int MAX = 42;
    public int previous = 0;
    public int next() {
        previous += 2; previous %= MAX;
        return previous;
    }
}
```

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

2.
```java
public class VectAdditioner {
    private Point sum = new Point();

    public void add(Point p) {
        sum.x += p.x; sum.y += p.y;
    }

    public Point getSum() {
        return sum;
    }
}
```

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 :

   ```java
   EvenNumbersGenerator eng = new EvenNumbersGenerator();
   eng.previous = 1;
   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 :

   ```java
   VectAdditioner va = new VectAdditioner();
   va.getSum().x = -12;
   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` :

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

   (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 : Aliasing, vérification de paramètres de constructeur

On commence avec la classe qui est sensée d'implémenter une date (oublions les classes java qui le font bien pour nous).

```java
public class SimpleDate {

    private int jour;
    private int mois;
    private int annee;

    public int getAnnee() {
        return annee;
    }

    public int getJour() {
        return jour;
    }

    public int getMois() {
        return mois;
    }

    public void setAnnee(int annee) {
        this.annee = annee;
    }

    public void setJour(int jour) {
        this.jour = jour;
    }

    public void setMois(int mois) {
        this.mois = mois;
    }

    public SimpleDate(int jour, int mois, int annee) {
        this.jour = jour;
        this.mois = mois;
        this.annee = annee;
    }
}
```

– Réécrire la méthode `toString` pour qu'elle retourne la date en format `jour/mois/annnee`.

Correction :

```java
@Override public String toString() {
    return String.format("%d/%d/%d", jour, mois, annee);
}
```

– Ajouter la méthode

```java
public boolean isValid(int jour, int mois, int annee)
```

qui vérifie si la date est correcte [1].

Correction :

```java
public static boolean isValid(int jour, int mois, int annee) {
    if (jour < 1)
        return false;

    switch (mois) {
        case 1, 3, 5, 7, 8, 10, 12:
            return jour <= 31;
        case 4, 6, 9, 11:
            return jour <= 30;
        case 2:
            if (annee % 400 == 0 || (annee % 4 == 0 && annee % 100 != 0))
                return jour <= 29;
            else
```

1. Rappelons que l'année est bissextile (le mois de février avec 29 jours) si l'année est divisible par 4 sans être divisible par 100 ou l'année est divisible par 400.

```
                    return jour <= 28;
        }
        return false;
    }

```

– Est-ce qu'il y a un modificateur qui manque dans la définition de `isValid` ?

Correction : Méthode n'utilise pas `this` donc il manque `static`.

– Le constructeur doit lancer l'exception `IllegalArgumentException` si la date proposée en argument n'est pas correcte.

Correction :

```

    public SimpleDate(int jour, int mois, int annee) {
        if (mois < 1 || mois > 12)
            throw new IllegalArgumentException("mois invalide");
        if (!isValid(jour, mois, annee))
            throw new IllegalArgumentException("jour invalide");
        this.jour = jour;
        this.mois = mois;
        this.annee = annee;
    }

```

La classe `SimpleDate` est utilisée par la classe `Employe` :

```
public class Employe {
    private String nom;
    private String prenom;
    private SimpleDate dateEmbauche;

    public Employe(String nom, String prenom,
     SimpleDate dateEmbauche) {
        this.nom = nom;
        this.prenom = prenom;
        this.dateEmbauche = dateEmbauche;
    }

    public String getNom() {
        return nom;
    }

    public String getPrenom() {
        return prenom;
    }

    public SimpleDate getDateEmbauche() {
        return dateEmbauche;
    }
}
```

Nous voulons que la classe `Employe` soit immuable (impossible de changer `nom`, `prenom` et `dataEmbauche` après la création d'instance).

– Est-ce que c'est le cas maintenant pour les trois champs ?

Correction : Oui pour `nom` et `prenom`. Mais `dateEmbauche` facilement modifiable après la création, comme ici :

```
Employe e1 = new Employe("Dupont", "Julien", new SimpleDate(3, 1, 2000));
System.out.println(e1.getDateEmbauche());
e1.getDateEmbauche().setAnnee(2021);
System.out.println(e1.getDateEmbauche());
```

– Sinon montrer comment changer un champ dans un objet `Employe` après la création.

– Est-ce que l'ajout de l'attribut `final` dans le(s) champ(s) résout le problème ?

Correction : Non.

– Modifier les classes `SimpleDate` et `Employe` pour qu'un `Employe` devienne immuable (mais il est interdit de supprimer les « `setters` » dans `SimpleDate`).

Correction : Ajouter la méthode

```
public SimpleDate copy(){
        return new SimpleDate(this.jour, this.mois, this.annee);
}
```

dans `SimpleDate` et modifier `getDateEmbauche` dans `Employe` :

```
  public SimpleDate getDateEmbauche() {
        return dateEmbauche.copy();
  }
```

– Tous les employés possèdent un seul et unique chef : Jules César (la date embauche le 14 février 44 av. J.-C). Ecrire la méthode

```
Employé getChef()
```

qui retourne ce chef unique. Mettez les modificateurs appropriés pour le champ `chef` et la méthode `getChef`.

Correction : Ajouter dans `Employe` :

```
private final static Employe chef =
            new Employe("Caesar", "Jules", new SimpleDate(12,7,-100) );
public static Employe getChef(){
        return chef;
}
```

Exercice 6 :

Le but d'exercice est de compléter la classe

```
public class PreciousStone {
    private String species; /* beryl */
    private String variety; /*  emerald aquamarine heliodor morganite etc */
    private BigDecimal refractiveIndex;
    private BigDecimal weight;
    private int hardness; /* entre 1 et 10 */
    private String  cleavage; /* basal, pinacoidal, cubic planar, octahedral, dodecahedral etc */
    private boolean inclusions;
    private int water; /* 1, 2, 3 */
}
```

La définition de la classe `PreciousStone` doit satisfaire les contraintes suivantes :

(1) Les valeurs de champs `species` et `variety` doivent être impérativement fournies à la création de l'instance.

(2) Les autres champs, appelons les « facultatifs » : `refractiveIndex`, `weight`, `hardness`, `cleavage`, `inclusions`, `water` possèdent les valeurs par défaut que l'instance prendra si d'autres valeurs ne sont pas spécifiées à la création de l'objet (respectivement les valeurs par défaut : 1.77, 0.3, 8, `"octahedral"`, `false`, 1). A la création d'instance de `PreciousStone` l'utilisateur peut choisir un sous-ensable quelconque de champs facultatifs à initialiser explicitement et pour d'autres champs facultatifs laisser les valeurs par défaut (cela donne 2^6 de possibilité de choix d'ensembles de champs à initialiser).

Je crois que c'est inutile de dire qu'écrire un constructeur séparé pour chaque ensemble possible de champs qu'on initialise explicitement à la création d'instance n'est pas envisageable.

Correction : Il faut un `Builder`. Et sans doute pas la peine de faire une solution complète en TP.

```java
public class PreciousStone {
    private String species; /* beryl etc */
    private String variety; /*  emerald aquamarine heliodor morganite etc */
    private BigDecimal refractiveIndex;
    private BigDecimal weight;
    private int hardness; /* entre 1 et 10 */
    private String cleavage; /* basal, pinacoidal, cubic planar, octahedral, dodecahedral etc */
    private boolean inclusions;
    private int water; /* 1, 2, 3 */

    private PreciousStone(Builder builder) {
        species = builder.species;
        variety = builder.variety;
        refractiveIndex = builder.refractiveIndex;
        weight = builder.weight;
        hardness = builder.hardness;
        cleavage = builder.cleavage;
        inclusions = builder.inclusions;
        water = builder.water;
    }

    public int getHardness() {
        return hardness;
    }

    public BigDecimal getRefractiveIndex() {
        return refractiveIndex;
    }

    public BigDecimal getWeight() {
        return weight;
    }

    public boolean isInclusions() {
        return inclusions;
    }

    public int getWater() {
        return water;
    }

    public String getCleavage() {
        return cleavage;
    }

    public String getSpecies() {
        return species;
    }

    public String getVariety() {
        return variety;
    }

    public static class Builder {
        private final String species; /* beryl */
        private String variety; /*  emerald aquamarine heliodor morganite etc */
        private BigDecimal refractiveIndex = new BigDecimal("1.77");
        private BigDecimal weight = new BigDecimal("0.3");
        private int hardness = 8; /* entre 1 et 10 */
        private String cleavage = "octahedral"; /* basal, pinacoidal, cubic planar, octahedral, dodecahedral etc */
        private boolean inclusions = false;
        private int water = 1; /* 1, 2, 3 */


        public Builder(String species, String variety) {
            this.species = species;
            this.variety = variety;
        }

        public Builder refractiveIndex(String val) {
            refractiveIndex = new BigDecimal(val);
            return this;
        }

        public Builder weight(String val) {
            weight = new BigDecimal(val);
            return this;
        }

        public Builder hardness(int val) {
            hardness = val;
            return this;
        }

        public Builder cleavage(String val) {
            cleavage = val;
            return this;
        }

        public Builder inclusions(boolean val) {
            inclusions = val;
            return this;
        }

        public Builder water(int val) {
            water = val;
            return this;
        }

        public PreciousStone build() {
            return new PreciousStone(this);
        }
    }
}
```

Exercice 7 : Nombres complexes

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)

```java
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 :

```java

public final class Complexe {

    /*
    * on pourrait presque mettre les attributs en public sans dégâts dans ce
    * cas (final), mais faire ainsi nous empêcherait dans des évolutions
    * futures de changer la représentation interne, donc nous allons quand-même
    * définir les getteurs.
    */
    public final double re, im;
    public final static Complexe I = new Complexe(0, 1);

    public double getRe() {
        return re;
    }

    public double getIm() {
        return im;
    }

    /*
    * Constructeur privé pour forcer à  utiliser les méthodes fabrique statiques
    */
    private Complexe(double re, double im) {
        this.re = re;
        this.im = im;
    }

    @Override
    public String toString() {
        return "(" + re + " + " + im + "i)";
    }

    /*
    * NB: il faudrait normalement redéfinir en même temps la méthode
    * hashCode() de façon cohérente (EJ 3 Item 11).
    */
```

```java
    @Override
    public boolean equals(Object autre) {
        if (autre == null || !(autre instanceof Complexe)) return false;
        Complexe autreComplexe = (Complexe) autre;
        return re == autreComplexe.re
            && im == autreComplexe.im;
    }

    public Complexe plus(Complexe autre) {
        return new Complexe(re + autre.re, im + autre.im);
    }

    public Complexe moins(Complexe autre) {
        return new Complexe(re - autre.re, im - autre.im);
    }

    public Complexe fois(Complexe autre) {
        return new Complexe(re * autre.re - im * autre.im, re * autre.im + im * autre.re);
    }

    public Complexe divisePar(Complexe autre) {
        if (autre.egale(ZERO))
            throw new ArithmeticException("Division by zero.");
        double d = autre.re * autre.re + autre.im * autre.im;
        double r = re * autre.re;
        double i = +im * autre.im;
        return new Complexe((r + i) / d, (r - i) / d);
    }

    public Complexe conjugue() {
        return new Complexe(re, -im);
    }

    public double module() {
        return Math.sqrt(re * re + im * im);
    }

    public double argument() {
        if (re == 0) {
            if (im == 0) return Double.NaN;
            else if (im < 0) return -Math.PI / 2;
            else return Math.PI / 2;
        }
        else if (re > 0) return Math.atan(im / re);
        else if (im >= 0) return Math.atan(im / re) + Math.PI;
        else return Math.atan(im / re) - Math.PI;
    }

    private static final Complexe ZERO = new Complexe(0, 0);

    private static final Complexe UN = new Complexe(1, 0);

    private static final Complexe I = new Complexe(0, 1);

    public static Complexe fromPolarCoordinates(double rho, double theta) {
        return new Complexe(rho * Math.cos(theta), rho = Math.sin(theta));
    }

    /*
    * Ne fait rien de plus que le constructeur, mais c'est plus équilibré de
    * mettre les deux modes de contsruction sur un pied d'égalité dans l'API.
    */
    public static Complexe fromRealAndImaginary(double re, double im) {
        return new Complexe(re, im);
    }
}

```

3. Améliorez l'encapsulation de votre classe, afin de permettre des évolutions ultérieures sans ń casser ż les clients/dépendents de celle-ci : en l'occurrence, les attributs doivent être privés

et des accesseurs publics [2] 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 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 [3] et le comportement des méthodes des opérations arithmétiques afin que le résultat soit enregistré dans l'objet courant (`this`), plutôt que retourné.

2. 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).

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