Université Paris Cité

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

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.
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.
5. On écrit `public` devant la déclaration d'une variable locale pour qu'elle soit visible depuis les autres classes.
6. Dès lors qu'un objet n'est plus utilisé, il faut penser à demander à Java de libérer la mémoire qu'il occupe.
7. La durée de vie d'un attribut statique est liée à celle d'une instance donnée de la classe où il est défini.

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
```

Exercice 3 : 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.
```
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.

Exercice 4 : Aliasing, vérification de paramètres de constructeur

On commence avec la classe qui est censée 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`.
- Ajouter la méthode

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

qui vérifie si la date est correcte [1].
- Est-ce qu'il y a un modificateur qui manque dans la définition de `isValid` ?
- Le constructeur doit lancer l'exception `IllegalArgumentException` si la date proposée en argument n'est pas correcte.

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

```java
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 `dateEmbauche` après la création d'instance).

- Est-ce que c'est le cas maintenant pour les trois champs ?
- 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 ?
- Modifier les classes `SimpleDate` et `Employe` pour qu'un `Employe` devienne immuable (mais il est interdit de supprimer les « `setters` » dans `SimpleDate`).

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.

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

```
Employe getChef()
```

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

Exercice 5 :

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). À la création d'instance de `PreciousStone` l'utilisateur peut choisir un sous-ensemble quelconque de champs facultatifs à initialiser explicitement et pour d'autres champs facultatifs laisser les valeurs par défaut (cela donne 2^6 possibilités de choix d'ensemble 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.

Exercice 6 : 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)

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 [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`.