TD et TP de CPOO n° 5 et 6 : Mécanismes et stratégies d'héritage, énumérations

I) Exercices

Exercice 1 : Personnes

```java
class Personne {
    private String nom;
    Personne(String nom) { this.nom=nom; }
    void presenteToi() { System.out.println("Je suis " + nom ); }
    void chante() { System.out.println("la-la-la"); }
}

class Enseignant extends Personne {
    private String matiere;
    Enseignant(String nom, String matiere) { super(nom); this.matiere=matiere; }
    void presenteToi() { System.out.println("Je suis " + nom + ",enseignant de "+ matiere); }
    void enseigne() { System.out.println(matiere + "is beautiful"); }
}

public class Test {
    public static void main (String[] args){
        Personne yan = new Personne ("Jurski");
        Enseignant isabelle = new Enseignant("Fagnot","CPOO5");
        Personne aldric = new Enseignant("Degorre","CPOO5");
        isabelle.chante(); yan.enseigne(); aldric.chante(); aldric.enseigne();
        Personne[] jury = { yan, isabelle, aldric, new Enseignant("Shen", "CPOO5"), (Personne) new
            Enseignant("Fantome", "CPOO5") };
        for(Personne p:jury) p.presenteToi();
    }
}
```

1. Lisez, comprenez et corrigez ce code, le cas échéant.
2. Dans la classe `Enseignant`, quels sont les attributs hérités, ajoutés ? Quelles sont les méthodes héritées, ajoutées, redéfinies ?
3. Comment fonctionne le constructeur de `Enseignant` ?
4. Qu'affiche le code ?

Exercice 2 : Liaison dynamique

Qu'affiche le programme suivant :

```java
class X {} class Y extends X {}

class A {
    void f(X y) { System.out.println("A et X"); }
    void f(Y y) { System.out.println("A et Y"); }
}

class B extends A {
    void f(X y) { System.out.println("B et X"); }
    void f(Y y) { System.out.println("B et Y"); }
}

class C extends B {
    void f(Y y) { System.out.println("C et Y"); }
}

public class Test {
    public static void main(String args[]) {
        A a = new C();
        Y x = new Y();
        a.f((X) x); // affichage n°1
        a.f(x); // affichage n°2
    }
}
```

II) Modélisation géométrique : limites de l'héritage et du sous-typage

Exercice 3 : Rectangle et carrés

Le problème décrit ci-dessous est aussi connu sous le nom du problème "cercle-ellipse". La relation entre un carré et un rectangle est en effet analogue à celle entre le cercle et l'ellipse. Plus d'informations ici : `https://en.wikipedia.org/wiki/Circle-ellipse_problem`.

1. Programmez une classe `Rectangle`. Un rectangle est un polygône à 4 côtés avec 4 angles droits. Dans un repère orthonormé, on peut le caractériser, de façon minimale, par
   — les coordonnées de son premier point (ex : pour rectangle $ABCD$, le point A),
   — l'angle de son "premier côté" (ex : l'angle du vecteur $\overrightarrow{AB}$)
   — et la longueur de ses deux premiers côtés (ex : AB et BC).
   Les attributs sont privés, accessibles par "getteurs". N'écrivez pas encore les "setteurs".
2. Tous les livres de géométrie élémentaire disent qu'un carré est un rectangle particulier, dont tous les côtés sont égaux.
   En POO, la relation "est un" se traduit habituellement par de l'héritage.
   Programmez donc la classe `Carre` qui étend `Rectangle` tout en garantissant que l'objet obtenu représente bien un carré.
3. Premier problème : un carré représenté par une instance de `Carre` contient plus d'attributs que nécessaire. Voyez-vous pourquoi ?
   Le problème de la taille en mémoire n'est en fait pas très grave. Ce qui est plus embêtant, c'est que la redondance peut induire des problèmes de cohérence. Voir ci-dessous.
4. Ajoutez maintenant les "setteurs" à la classe `Rectangle` (notamment `setLongueur` et `setLargeur`, modifiant respectivement la longueur et la largeur, sans toucher aux autres propriétés du rectangle).
5. Si dans une méthode on fait :

```
1 Carre c = new Carre(/*ax = */ 0, /*ay = */ 0, /* angle = */ 0, /* cote */ = 3);
2 c.setLongueur(10);
```

   l'objet `c` correspond-il toujours à la modélisation d'un carré ?
   L'objet `c` est-il pourtant encore de type `Carre` ?
6. Pour corriger ce problème, redéfinissez (`@Override`) les setteurs dans la classe `Carre` de sorte à préserver l'invariant "cet objet représente un carré". Une possibilité : modifier la longueur modifie aussi la largeur, et vice-versa.
7. Dans la documentation de la classe `Rectangle` (p. ex. : dans la javadoc), il serait raisonnable d'écrire comme spécification pour la méthode `setLongueur`, une phrase comme "modifie la longueur de ce rectangle sans modifier ses autres propriétés" (et une phrase similaire pour `setLargeur`).
   Si une méthode contient les instructions suivante :

```
1 Rectangle r = new Carre(/*ax = */ 0, /*ay = */ 0, /* angle = */ 0, /* cote */ = 3);
2 r.setLongueur(10);
```

   quelle sera alors la largeur de `r` ? La spécification décrite plus haut est-elle alors respectée ?
8. Dire que `Carre` hérite de `Rectangle` n'a pas l'air de fonctionner bien. Mais peut-on faire le contraire ? Après tout, un rectangle utilise une grandeur en plus, par rapport au carré.
   En vous inspirant de ce qui précède, montrez que ça ne marche pas non plus.

Dans cet exercice, on vient de montrer que, bien qu'un carré, en tant qu'entité mathématique figée, soit un rectangle particulier, cette inclusion n'est plus valable quand on parle de carrés et de rectangles en tant qu'objets modifiables préservant leur identité de carré ou de rectangle.

Dans ce cadre, il est donc illusoire de vouloir que `Carre` *soit sous-type de (et a fortiori sous-classe de)* `Rectangle`*. On verra dans la suite qu'on peut obtenir un sous-typage satisfaisant en écrivant des types* `Carre` *et* `Rectangle` *immuables.*

Exercice 4 : Quadrilatères

Cet exercice est, a priori, indépendant du précédent. Pour un programme propre, il est recommandé de repartir de zéro pour son écriture.

On vient de voir dans l'exercice précédent que la spécification d'un objet modifiable est facilement mise à mal par le sous-typage. Ainsi, dans cet exercice, on n'écrira pas de mutateurs (pour de vraies classes immuables, attendez la suite !).

1. On veut écrire des classes pour les formes suivantes : quadrilatère, trapèze, parallélogramme, losange, rectangle, carré. Vous pouvez consulter `https://fr.wikipedia.org/wiki/Quadrilatère` pour plus de détails.

   Dessinez le graphe de sous-typage idéal. Est-ce qu'il sera possible de le réaliser par uniquement des classes, en matérialisant le sous-typage par de l'héritage ? Pour quelle raison ?

   Quelles solutions peut-on envisager pour régler ce problème ?

   Pour l'instant nous nous limiterons aux classes `Quadrilateral`, `Parallelogram`, `Rectangle` et `Square`. Vérifiez que cela suffit pour éviter le problème soulevé.

2. Nous allons demander en plus que nos figures implémentent toutes l'interface

```java
interface Shape2D {
    double perimeter();
    double surface();

    /**
     *  Méthode servant juste au test. Ne doit pas servir dans le programme final.
     *  @return true si la figure a bien les propriétés qu'elle prétend avoir
     */
    default boolean checkInvariants() { return true; }
}
```

   Écrivez la classe `Quadrilateral`. Pour les sommets, vous pouvez utiliser la classe `Point2D.Double` (vous utiliserez sans doute ses méthodes `double distance(Point2D pt)` et `double distanceSq(Point2D pt)`). Pour la surface d'un quadrilatère $ABCD$, vous pouvez utiliser la formule suivante :

$$A = \frac{1}{2}\left(x_1 y_2 - x_2 y_1\right)$$

   où $\vec{AC} = (x_1, y_1)$ et $\vec{BD} = (x_2, y_2)$. Il s'agit d'une aire algébrique (peut être négative). Vous pouvez prendre la valeur absolue si ça vous arrange (mais la formule devient fausse pour un quadrilatère croisé).

3. Passez les attributs en `private final` et ajoutez les getteurs pour les sommets.

4. Écrivez la classe `Parallelogram` (avec l'héritage qui va bien). Vous devez garantir qu'à la construction, les côtés opposés sont parallèles (pour vérifier si deux vecteurs sont parallèles, il suffit de vérifier $x_1 y_2 - x_2 y_1 = 0$) ; une façon de garantir le parallélisme est de ne demander que 3 sommets et de calculer automatiquement la position du 4e.

N'hésitez pas à créer des méthodes auxiliaires pour éviter d'écrire du code répétitif ! Si vous pensez qu'elles seront utiles dans les sous-classes, celles-ci devront être de visibilité `protected`, sinon `private`.

Redéfinissez la méthode `checkInvariants()` afin qu'elle renvoie `true` si et seulement si la propriété d'être un parallélogramme est effectivement vérifiée.

5. Les getteurs des sommets de votre classe `Quadrilateral` utilisent-ils, comme type de retour, le type `Point2D.Double` ou directement des valeurs de type `double` ?

   Voyez-vous en quoi utiliser `Point2D.Double` menace-t-il la préservation de l'invariant ?

   Si vous avez le temps, corrigez vos getteurs (sinon, ça n'empêche pas de continuer l'exercice). Solutions possibles :
   - retourner des `double`, plutôt qu'une référence vers l'instance de `Point2D.Double` encapsulée dans la figure (il faut alors 2 getteurs par sommet)
   - retourner une copie de l'instance de `Point2D.Double` plutôt qu'une référence vers l'instance encapsulée
   - ne pas utiliser `Point2D.Double` du tout → remplacer par des `double` ou bien par une classe `Point` de votre cru qui serait immuable.

6. Ne voyez-vous pas un problème similaire avec les constructeurs : que se passerait-il si un utilisateur instanciait un `Point2D.Double`, gardait sa référence dans une variable `s`, puis le passait en argument du constructeur d'un `Parallelogram` et enfin, modifiait le point référencé par la variable `s` ?

   Comment corriger ce problème ?

7. Écrivez les classes `Rectangle` et `Square`, en respectant la même hygiène que pour `Parallelogram`. Attention : le rectangle a un invariant de plus que le parallélogramme : les angles doivent rester droits ; et le carré a encore un invariant supplémentaire : tous les côtés ont la même longueur.

   Notez qu'à ce point, on ne doit plus avoir directement accès aux attributs et que la seule façon d'accéder aux sommets, c'est via les accesseurs hérités (`super.getA()`, `super.setB(...)`, ...) et les éventuelles méthodes auxiliaires `protected`.

Arrivé à la fin de cet exercice, on a normalement des classes pour représenter différentes figures, avec les garanties suivantes :

- *toute instance directe des classes programmées représente bien à tout moment une figure du type donné par le nom de la classe (ex : une intance de `Carre` représente un carré)*
- *les méthodes `perimeter()` et `surface()` retournent bien respectivement le périmètre et l'aire de la figure.*

Mais ces garanties ne valent que pour les instances directes de ces classes. Il est encore possible de tout "casser" en créant des mauvaises sous-classes. Cela pourra être règlé en ajoutant le mot-clé `final` devant la déclaration de la classe qui ne doit pas être extensible, mais ce n'est pas si simple : dans cet exercice on eu besoin de l'héritage, par exemple pour passer de `Parallelogram` à `Rectangle`. Il faudra donc "ruser" (à suivre...).

On commence d'ailleurs à distinguer ce qui est nécessaire pour écrire une classe immuable :

- *ne pas permettre la modification des attributs ;*
- *si certains attributs référencent des objets mutables, faire en sorte qu'aucune référence vers ces objets ne puisse exister à l'extérieur de la classe (faire des copies défensives) ;*
- *empêcher de créer des sous-classes.*

Exercice 5 : Cylindres

Nous voulons maintenant construire des cylindres (solides constitués de deux bases parallèles superposables et d'une "surface cylindrique" constituée de lignes droites parallèles joignant les deux bases). Ainsi, en ce qui nous concerne, un cylindre se caractérise par sa base (une forme 2D telle que décrite dans l'exercice précédent) et sa hauteur.

Sur de tels cylindres, nous voudrions en outre calculer la surface (= 2 fois la surface de la base + hauteur * périmètre de la base) et le volume (= surface de la base * hauteur).

1. Si nous écrivions une classe `QuadriCylinder`, pour un cylindre à base quadrilatère, qui serait sous-classe de `Quadrilateral` (avec attribut ajouté `height`, ainsi que les méthodes ajoutées `surface()` et `volume()`), est-ce qu'il serait possible d'obtenir une classe `SquareCylinder` pour un cylindre à base carrée qui serait sous-type de `Square` et de `QuadriCylinder` ?

   Est-ce que cette classe `QuadriCylinder` était une bonne idée de toute façon ? (argumentez)

2. Nous allons nous y prendre autrement et utiliser la composition : un cylindre n'est pas une forme 2D "améliorée", mais une forme 3D qui possède une base, qui est une forme 2D, ainsi qu'une hauteur.

   Écrivez une telle classe, avec les méthodes surface et volume demandées.

3. Peut-on maintenant écrire des sous-classes de cylindres, en fonction de la forme de leur base ? Si oui, faites-le.

   Faites attention à ce qu'il soit impossible de changer le type de base pendant la vie d'un tel objet (on ne peut pas modifier la base d'un cylindre à base carrée de telle sorte à ce qu'il devienne un cylindre à base circulaire, par exemple).

III) Pause documentation

Lisez le complément de cours sur l'immuabilité (cf. Moodle) avant de passer à la suite.

IV) Modélisation géométrique (suite)

Exercice 6 : Rectangles et carrés, une solution ?

Comme le problème de l'exercice **??** est qu'on ne peut pas concilier à la fois l'invariant "cette figure est un XXX" et la spécification des mutateurs héritée du supertype, une version immuable des figures ne devrait être plus robuste.

Évidemment, les mutateurs fournissaient une fonctionnalité utile. Pour les remplacer, il est possible d'écrire des méthodes retournant un nouvel objet identique à `this`, sauf pour la propriété qu'on souhaite "modifier". Exemple : `rect.withLongueur(15)` retourne un rectangle identique à `rect` mais dont la longueur est 15 ; en revanche, `carre.withLongueur(15)`, pour préserver l'indépendance des propriétés largeur et longueur, retourne un rectangle et non un carré.

À faire :

1. Écrivez les classes immuables `RectangleImmuable` et `CarreImmuable` sous-classes de `Rectangle` (classe abstraite fournissant les fonctionnalités communes, sans mutateur, mais avec les opérations `withXXX()` en tant que méthodes abstraites). Notez que les méthodes de "modification" de `CarreImmuable` respectent automatiquement l'invariant du carré car `this` n'est pas modifié.
2. Ci-dessus, pour empêcher la création de sous-types mutables, `RectangleImmuable` n'est pas extensible ; pour cette raison, `CarreImmuable` n'en n'est pas un sous-type.
   Cependant la technique des classes scellées permet, pour une classe donnée, de définir une liste fermée de sous-types et donc de garantir l'immuabilité du supertype. Utilisez cette technique pour définir des types immuables `Rectangle` et `Carre` avec `Carre` sous-type de `Rectangle` (imbriquez vos déclarations dans une classe-bibliothèque `FormesImmuables`).

Exercice 7 : Quadrilatères

1. Modifiez les classes de l'exercice **??** pour les rendre immuables.
2. Ajoutez-y des méthodes pour "déplacer" les différents sommets. Comme les figures ne sont pas modifiables, ces méthodes retournent donc une nouvelle figure.
3. Ajoutez des méthodes de conversion d'une figure à une autre. Par exemple : `public Rectangle toRectangle()`.
   Ces méthodes peuvent être déclarées en tant que méthodes abstraites très haut dans l'arbre d'héritage (classe `Figure`).
   Évidemment, certaines conversions n'ont pas de sens (p. ex : convertir un parallélogramme quelconque en carré). Ce qu'on peut faire, c'est vérifier que la figure `this` représente effectivement une figure du type vers lequel on veut la convertir (pour le carré : vérifier angle droit et 4 côtés égaux), et retourner `null` [1] quand la vérification échoue.
   Remarque : à cause de l'arithmétique flottante, la précision n'est pas absolue. Il faut donc se permettre une marge d'erreur, sinon on ne pourrait presque jamais convertir un rectangle en carré. Cette marge peut être paramétrée par un attribut statique de `Figure` ou bien être passée en paramètre des méthodes de conversion.

1. On peut explorer d'autres solutions plus "propres" : la classe `Optional`, ou bien lancer des exceptions.

V) Encore une pause

Lisez le sous-chapitre “Énumérations” du cours (14 transparents) avant de passer à la suite.

VI) Cartes (mini-projet)

Exercice 8 : Jeu de cartes – Modélisation

Le jeu de cartes classique se compose de 54 cartes : 52 cartes dans 4 familles correspondant à 4 enseignes différentes (pique, cœur, carreau, trèfle), chacune composée de 13 valeurs (as, deux, trois, ... neuf, dix, valet, dame, roi), ainsi que 2 jokers.

1. Pour le jeu à 52 cartes (sans joker), proposez une modélisation objet utilisant les énumérations, permettant de représenter toutes les cartes de ce jeu de cartes. Pour une carte donnée, il faut être capable de récupérer :
   — son enseigne (pique/cœur/carreau/trèfle, appelée souvent “couleur”, mais pour éviter les confusions, nous éviterons ce terme)
   — sa valeur
   — sa couleur (rouge ou noir)
   — l’information si cette carte une figure (valet, dame, roi) ou pas
   Essayez de faire en sorte de placer le plus d’implémentation possible au sein des `enum`, quitte à ce que les méthodes des cartes se contentent d’appeler celles des `enum`.
2. Adaptez vos classes (et peut-être même la structure de sous-typage) afin que les classes directement instanciables soient immuables.
3. Proposez une modélisation pour ajouter les jokers (il faut que les cartes “normales” soient instances d’un même type `CarteNormale` dont les jokers ne sont pas... mais il faut aussi que toutes les cartes, jokers compris, soient instances du type `Carte`).
4. Rendez le type `Joker` immuable.
   Maintenant, `CarteNormale` et `Joker` sont immuables. Mais est-ce que `Carte` l’est ? Si ce n’est pas le cas, corrigez ce problème (faites une classe scellée).
5. Programmez des méthodes statiques permettant de générer : un jeu de 52 cartes standard (retourné sous la forme d’une collection), un jeu de 32, un jeu de 54 (avec jokers),
6. Même question pour générer une collection contenant plusieurs exemplaires d’un jeu de carte (par exemple, pour jouer au Rami, il faut deux jeux de 54).

Exercice 9 : Jeu de cartes – 8 américain

En utilisant les cartes programmées à l’exercice **??**, programmez le jeu du 8 américain (cf. Wikipédia), ancêtre du jeu Uno. Programmez juste la version présentée comme “version minimale”. Une version simple à 2 joueurs conviendra (humain contre humain, prévoyez un code simple pour désigner les cartes en ligne de commande ; intégrez judicieusement la reconnaissance de ce code dans les classes et énumérations de l’exercice **??**).

VII) Une pause décoration

Lisez le sous-chapitre “Discussions” du cours (dans la partie Héritage) avant de passer à la suite.

VIII) Encodage et compression transparente (mini-projet)

Exercice 10 : Stockage

On souhaite écrire un programme simple de gestion du personnel. Vous êtes responsable de l'aspect stockage des données et on vous a confié comme mission d'écrire une classe permettant de lire et d'écrire la liste du personnel depuis/vers un fichier. Pour simplifier, on suppose que la liste du personnel est une (très longue) chaine de caractère et qu'on la lit ou écrit en entier d'un seul coup.

1. Écrivez une classe `Stockage`. Cette classe a un constructeur prenant un paramètre (le nom du fichier) et doit permettre de
   — lire tout le contenu et le retourner sous forme d'une `String` ;
   — effacer tout le contenu et le remplacer par une `String` passé en paramètre.
   On stockera le nom du fichier comme un attribut privé, sans aucun "getter" ou "setter". On pourra utiliser par exemple `Files.readAllBytes`, `Files.write` et `Path.get`, de la bibliothèque `java.nio.file`. Testez votre code avec un fichier.
2. On vous informe que le programme devra supporter plusieurs types de stockage possibles, par exemple dans un fichier, sur le réseau, en mémoire, etc. On vous demande d'étendre votre code pour gérer cela de façon transparente. Modifiez votre code pour que `Stockage` devienne une classe abstraite. Votre code précédent devient maintenant la classe `StockageFichier` qui hérite de `Stockage`. Implémentez un autre type de stockage en mémoire : `StockageMemoire` stocke le contenu dans un attribut privé de type `String`. Son constructeur prend en paramètre le contenu.

Exercice 11 : Chiffrement

Le projet avance bien et votre code de stockage fonctionne bien. Soudain, on vous fait remarquer que stocker des données sensibles comme des salaires ou adresses en clair n'est pas une bonne pratique de sécurité. Vous décidez donc de chiffrer les données qui sont écrites et de déchiffrer celles qui sont lues. Malheureusement, vous ne pouvez plus toucher à l'interface `Stockage` si tard dans le projet. Vous pourriez ajouter le chiffrement et déchiffrement à chaque implémentation de `Stockage` mais cela duplique beaucoup de code. Le patron décorateur vous paraît être un bon moyen de se sortir de cette situation.

1. Créez une classe abstraite `StockageDecorateur` qui hérite de `Stockage`. Celle-ci contient un attribute de type `Stockage` (composition) et son constructeur prend en paramètre un objet de type `Stockage` qu'elle stocke dans cet attribut. Les méthodes de lecture et écriture se contentent d'appeler les méthodes correspondantes de l'objet stocké.
2. Créez une classe `StockageChiffre` qui hérite de `StockageDecorateur`. Ajoutez deux méthodes `chiffrer` et `dechiffrer` qui chiffre et déchiffre (ainsi `dechiffrer(chiffrer(x))` doit retourner `x`). La méthode de lecture doit maintenant lire depuis l'objet décoré puis déchiffrer ce qu'elle a lu. La méthode d'écriture doit chiffrer le contenu puis l'écrire avec l'objet décoré. On pourra utiliser comme méthode d'encodage une simple rotation des caractères comme ROT13 (cf Wikipédia). Vérifier que votre code fonctionne sur l'exemple suivante :

```
1 Stockage mem = new StockageMemoire("");
2 Stockage s = new StockageChiffre(mem);
3 s.ecrire("Monsieur Smith, espion, 100000 euros");
4 System.out.println("Contenu chiffre: " + mem.lire());
5 System.out.println("Contenu déchiffré: " + s.lire());
```

Exercice 12 : Compression

Maintenant que le programme est complet, vous vous apercevez que la liste du personnel peut être grande. Cela pose un problème lorsque vous devez la transférer sur un réseau par exemple, qui est beaucoup plus lent qu'un disque dur. Une façon simple de régler ce problème est de compresser les données avant de les envoyer. Heureusement, grâce aux décorateurs, vous pouvez faire cela facilement.

Remarque : dans cet exercice, afin de simplifier le code, on n'utilisera pas une vraie méthode de compression, il s'agit simplement de voir le principe. Si vous souhaitez faire de la vraie compression, vous pouvez essayer d'utiliser `DeflaterOutputStream`, `InflaterInputStream` et consorts.

1. Créez une classe `StockageCompresse` qui hérite de `StockageDecorateur`. Ajoutez deux méthodes `compresser` et `decompresser` qui compressent et décompressent. La méthode de lecture doit lire depuis l'objet décoré puis décompresser ce qu'elle a lu. La méthode d'écriture doit compresser le contenu puis l'écrire avec l'objet décoré. Au lieu de "compresser", on va en fait encoder le contenu de Base64, vous pouvez utiliser la classe `Base64` de `java.util`, notamment `Base64.getEncoder()` pour "compresser" et `Base64.getDecodeR()` pour "décompresser". Vérifier que votre code fonctionne sur l'exemple suivante :

```
Stockage mem = new StockageMemoire("");
Stockage chif = new StockageChiffre(mem);
Stockage s = new StockageCompresse(chif);
s.ecrire("Monsieur Smith, espion, 100000 euros");
System.out.println("Contenu compressé puis chiffré: " + mem.lire());
System.out.println("Contenu compressé: " + chif.lire());
System.out.println("Contenu: " + s.lire());
```

2. Dans votre `main`, ajoutez un autre test qui écrit le même contenu, mais changez l'ordre des décorateurs pour chiffrer avant de compresser. Obtenez-vous le même résultat ? Que peut-on en conclure sur les décorateurs ? Selon vous, vaut-il mieux chiffrer avant de compresser ou compresser avant de chiffrer ?