Date : Octobre 2020
Enseignant : Marie-Laure Nivet
Diplôme : L3 SPI parcours Informatique
Nom de l'UE : Programmation orientée objet, langage Java
Type de document : TP

TP – Classes, Types Enum, Héritage, Interfaces, Exceptions

Objectifs

L'objectif de ce TP est de voir dans quelle mesure les concepts de la POO vus ou revus jusqu'à présent sont bien maîtrisés et d'en introduire certains non vus en cours : les types Enum. Si ce n'est pas le cas, vous êtes invités à revoir les cours 1, 2, 3 et 4 disponibles en téléchargement sur l'ENT, à poser des questions et à revoir les exercices effectués jusqu'alors.
Remarques : Comme habituellement ce sujet est bien évidemment sujet à interprétation (Il a été retouché quelque peu tardivement ;-))… N'hésitez pas à poser des questions si certaines choses ne vous paraissent pas claires et/ou discutables… Nous en discuterons !

Modalités de rendu

Vous devez travailler sur ce TP en programmation par binôme (ou pair programming cf. https://fr.wikipedia.org/wiki/Programmation_en_binome). Essayez de jouer vraiment le jeu, pour ceux qui ont une connexion qui le permet. Le but n'est pas d'aller le plus vite possible en scindant le travail en deux et en codant chacun de son côté en mettant en commun le travail à la fin, mais il est de coder réellement à deux : un qui observe (partage d'écran), fait de la critique constructive et vérifie l'adéquation au cahier des charges ; l'autre qui code. Les rôles s'inversent toutes les demi-heures par exemple.
Vous devrez rendre ce TP (uniquement *.java) lundi 14 Décembre 23h59 dernier délai. Il devra être disponible sur un dépôt Git dont le lien sera renseigné dans le tableau disponible dans la zone Fichier du canal réservé au cours.

Exercice 1 : Interfaces (extrait d'un examen)

On vous propose la définition d'interface[1] suivante :

```
package tp2;
interface EstComparable{
        int compareA(Object o);
}
```

ou `x.compareA(y)` doit retourner –1 si x est inférieur à y, 0 s'il sont égaux et 1 sinon.

[1] En fait, pour les petits curieux, cette interface s'inspire (pour ne pas dire singe !) l'interface Comparable du JDK. Vous en trouverez la description à l'adresse suivante https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/Comparable.html : Nous ne l'utilisons pas pour l'instant car il faut avoir en premier lieu vu le cours sur la généricité pour en comprendre toutes les subtilités avant d'en profiter pleinement ! Nous attaquerons ce cours la semaine prochaine…

Définissez et écrivez une classe `MonTableau` qui implémente `EstComparable` et dont les instances se comportent comme des tableaux d'entiers. La comparaison sur les instances de la classe `MonTableau` se fera sur la base de la somme des éléments du tableau. Par exemple :

```
int[] a = new int[] {1,2,3,4}
int[] b = new int[] {-1,2,-3,4,5};
MonTableau m1=new MonTableau(a);
MonTableau m2=new MonTableau(b);
System.out.println(m1.compareA(m2)); //Affiche 1,
//car 1+2+3+4 > -1+2-3+4+5
```

Attention si l'objet passé en paramètre n'est pas comparable car Null ou d'une classe ne permettant pas d'effectuer la comparaison avec une instance de MonTableau, il y aura levée d'exception, à vous de voir la ou lesquelles.
Ecrivez le code de la classe `MonTableau` tel que le code précédent puisse être exécuté.

Exercice 2 : Modélisation de citernes

Question 1 : On vous demande de créer une classe `Citerne` pour modéliser des cuves contenant divers liquides. Ce problème est illustré par la figure, page suivante.

- Une cuve est identifiée par un numéro unique (son numéro d'ordre de création par exemple) attribué automatiquement, qui permet d'assurer son unicité de référencement. Une fois ce numéro attribué il ne doit plus pouvoir être modifié.

- A sa construction on doit spécifier sa capacité, donnée en mètres cubes (un entier), une fois attribuée cette capacité ne pourra plus être modifiée. La capacité maximum des cuves constructible est de 20 000 m^3. Évidement une cuve ne doit pas pouvoir être construite sur la base d'une capacité négative ou nulle, vous gérerez cela en faisant lever à vos constructeurs des Exceptions lorsque cela sera nécessaire. A vous de choisir le type d'exception levée et d'éventuellement créer une nouvelle classe d'Exception si cela vous parait nécessaire.

- Toutes les cuves construites peuvent recevoir seulement trois types de liquide : de l'eau dont la température de conservation idéale est 10°, du vin dont la température de conservation idéale est 15°, de l'huile dont la température de conservation idéale est 9°. On spécifiera à la construction quel est le type de liquide destiné à être stocké dans la cuve. Il pourra être modifié ultérieurement, sous réserve que la citerne ait subi au préalable une opération de nettoyage complète (`nettoyage`). Le type de liquide sera modélisé par un type énuméré (cf. https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html).

- Lors de la construction de la cuve on conserve sa date de mise en service, c'est-à-dire la date système à laquelle l'instance de Citerne a été créée. On ne s'intéressera qu'aux données concernant l'année, le mois et le jour (cf : https://docs.oracle.com/javase/tutorial/datetime/iso/date.html). L'heure importe peu. Recherchez les classes Dates mises à votre disposition dans le JDK et choisissez celle qui vous parait la plus adaptée.

- On doit pouvoir via la méthode `plusAncienne` pouvoir retourner entre deux références de citernes passées en paramètre la plus ancienne des deux, c'est-à-dire celle ayant la date de mise en service de cuve la plus ancienne.

- À tout moment on désire pouvoir connaître le contenu d'une citerne en termes de quantité de liquide stockée, de type de liquide stocké et de date de mise en service.

- À tout moment on désire pouvoir connaître le nombre de citerne qui ont été construites.
- On désire pouvoir `ajouterLiquide` du liquide dans la cuve. Cette quantité pourra être exprimée par un nombre entier de mètres cube ou par la donnée d'un pourcentage (entre 0 et 1) de remplissage par rapport à la capacité initiale de la cuve. Évidement si la cuve est trop pleine pour recevoir tout le liquide on la rempliera à son maximum et on lèvera une exception signalant la quantité de liquide en dépassement. A vous de choisir en justifiant ce choix, le type de l'exception levée. Si la cuve vient d'être nettoyée et que le nouveau liquide assigné n'est pas encore spécifié il y aura également une levée d'exception.
- On désire également pouvoir vider la cuve `enleverLiquide` de tout ou partie de son contenu (soit par la donnée d'un cubage, soit par la donnée d'un pourcentage). De la même façon si la cuve ne contient pas assez de liquide par rapport à la quantité demandée au niveau du retrait, il devra y avoir levée d'exception signalant la quantité de liquide manquante pour satisfaire à la demande. Mais malgré tout la cuve sera dans ce cas intégralement vidée.
- Deux cuves seront considérées comme égales si elles ont la même capacité, la même date de mise en service, stockent le même type de liquide et sont remplies avec le même volume.
- A l'affichage d'une citerne on devra obtenir les informations suivantes :

```
Citerne n°1, Vin, capacité : 13 m3, mise en service : 2019-10-22, volume occupé : 9.5
```

Question 2 : Les Citernes doivent pouvoir être comparées les unes aux autres au sens du volume effectif de liquide contenu puis de leur capacité. Pour ce faire on vous demande de ré-utiliser l'interface `EstComparable` présentée à l'exercice 1 et de la faire implémenter à votre classe Citerne. Une Citerne c1 sera considérée inférieure à la citerne c2 si son volume effectif contenu est inférieur au volume contenu dans c2, et pour un volume égale c1 sera inférieure à c2 si sa capacité l'est.

Question 3 : Écrivez un code de test pour votre classe `Citerne, TestCiterne` possédant une méthode `main` démontrant son bon fonctionnement, c'est-à-dire que vous veillerez à invoquer au moins une fois toutes les méthodes implémentées. Vous veillerez à utiliser les assertions.

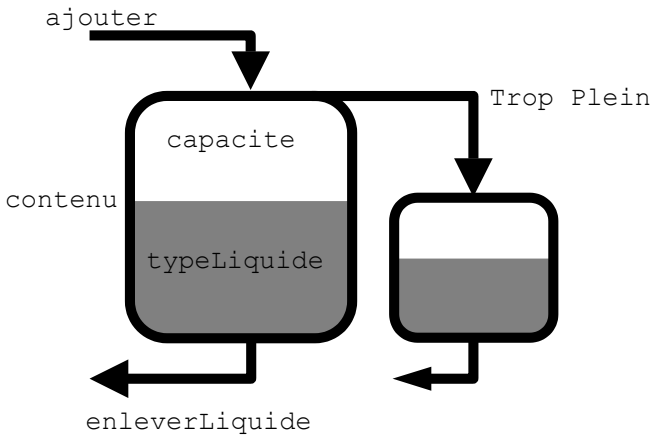
Question 4 : On souhaite maintenant pouvoir représenter des citernes sécurisées, classe `CiterneSecurisee`.

- C'est à dire des citernes possédant une cuve de trop plein. A vous de réfléchir à l'éventuelle nécessité de créer une Classe `TropPlein`.
- Ce trop plein doit permettre à la cuve de se décharger lorsque les opérations d'ajout de liquide dépassent la capacité de la cuve principale sans pour autant refuser l'opération d'ajout dans sa totalité.
- Un message d'alerte devra être affiché sur la console d'erreur lorsque la cuve principale déborde (se déchargeant dans la cuve de trop plein) ainsi que lorsque la cuve de trop plein est à moitié pleine. Charge alors au technicien de faire les opérations de vidanges nécessaires comme sur une cuve classique.

- L'opération de connexion de la cuve de trop plein à la cuve principale doit se faire obligatoirement à la construction de cette dernière. Cependant cette cuve de trop plein peut être modifiée à tout moment à la condition qu'il y en ait toujours une. Dans le cas où la cuve de trop plein renseignée ne correspondait pas aux attentes, une cuve par défaut dimensionnée à 10% de la capacité de la cuve principale serait automatiquement créée et assignée.

- La représentation textuelle d'une citerne sécurisée devra faire apparaître l'état et les caractéristiques de la cuve de trop plein.

- Une cuve de trop plein, elle, ne possède pas de trop plein… Sinon nous n'en finirions pas !



Donnez le code permettant de modéliser ce problème. N'oubliez pas de redéfinir toutes les méthodes importantes héritées de la classe Objet…
La comparaison entre deux Citernes Sécurisées se fera sur la comparaison combinée de la citerne principale et de son trop plein en cumulant les capacités et volumes.

Question 5 : Complétez vos classes pour mettre en place le clonage en profondeur des `CiterneSecurisee`.

Question 6 : Complétez votre classe `TestCiterne` pour démontrer le bon fonctionnement de votre classe `CiterneSecurisee`, c'est-à-dire que vous veillerez à invoquer au moins une fois toutes les méthodes implémentées. Vous veillerez par ailleurs à utiliser les assertions.