TD de programmation orientée objet en Java
Les classes

Exercice 1 : Personnes

Créer une classe `Personne` qui reprend, en partie, l'exemple vu en cours et qui permet de définir des objets représentant des personnes. Une personne est décrite par son nom, son prénom, son âge et son sexe. Votre classe doit proposer :

- Un constructeur par défaut qui ne prend aucun paramètre et qui permet de créer le fameux John Doe (on supposera que ce monsieur a 30 ans).
- Un constructeur qui prend en paramètre toutes les informations (nom, prénom, âge et sexe) et crée l'objet correspondant convenablement.
- Un ensemble d'accesseurs (ou getters) qui permettent de récupérer les valeurs des différents attributs de l'objet (ex. une méthode `getName()` qui permet de connaître le nom de la personne et ainsi de suite).
- Une méthode `sameLastName(Personne p)` qui prend en paramètre un deuxième objet de type `Personne` et qui permet de savoir si les deux personnes ont le même nom de famille.
- Une méthode `oldest(Personne p)` qui compare la personne appelante avec la personne fournie en paramètre et retourne celle qui est la plus âgée.

Exercice 2 : Nombres complexes

Un nombre complexe [1] z se présente généralement en coordonnées cartésiennes $z = a + bi$ où a est la partie réelle de z et b sa partie imaginaire (a et b sont deux réels et $i^2 = -1$). Créer une classe `Complexe` qui permet de décrire des nombres complexes définis sous cette forme. La classe doit fournir un constructeur convenable pour instancier ces nombres correctement. Elle doit également fournir des méthodes permettant de retourner pour un nombre complexe :

- Sa somme avec un deuxième nombre complexe passé en paramètre.
- Son produit avec un deuxième nombre complexe passé en paramètre.
- Son conjugué.
- Son module.
- Son carré.

Exercice 3 : Nombres rationnels arbitrairement longs

Un nombre rationnel [2] est un nombre qui peut s'exprimer comme le quotient de deux entiers relatifs : $\frac{num}{denom}$ (avec $denom \neq 0$). Créer une classe `BigRationnel` qui définit le type d'objets rationnels arbitrairement longs (c.à-d. le numérateur et le dénominateur peuvent avoir de très grandes valeurs et doivent être représentés sous forme de `BigInteger`). Votre classe doit proposer :

- Un constructeur adéquat qui prend en paramètre le numérateur et le dénominateur du rationnel et s'assurera que ce dernier est non nul avant de créer l'objet (faute de quoi il générera une exception avec l'instruction `throw new ArithmeticException("division par zéro");`).
- Des méthodes correspondant aux opérations usuelles sur les rationnels, à savoir :
  - L'addition.
  - La soustraction.
  - La multiplication.
  - La division (attention à la division par zéro).
- Un ensemble de méthodes de classe `valueOf()` (dont la signature est `public static BigRationnel valueOf(<TypeParamètre> <nomParamètre>)`). Chacune de ces méthodes prend en entrée un paramètre d'un type donné (ex. `String`, `int`, `BigInteger`, etc.) et renvoit sa représentation sous forme d'un `BigRationnel`.
- Les constantes `BigRationnel.ZERO` et `BigRationnel.ONE` qui représentent respectivement les entiers 0 et 1 (de la même manière que `BigInteger.ZERO` et `BigInteger.ONE`).

Exercice 4 : Digicode

Version modifiable

Créer une classe qui s'appelle `DigicodeM` et qui permet de modéliser un digicode [3]. Concrètement, un digicode contient, entre autres, les attributs suivants :

- `combination` qui est un tableau de caractères (`char []`) contenant le code de déverrouillage ;
- `locked` de type `boolean` qui contient l'état du digicode (verrouillé/déverrouillé).

Deux interactions sont possibles avec un digicode et ce grâce aux deux méthodes suivantes :

- `isLocked()` qui permet de savoir s'il est verrouillé ou pas ;
- `pushButton(char c)` qui permet d'"appuyer" sur l'une des touches du pavé numérique du digicode (nous supposons ici que tous les caractères sont acceptables). Si l'utilisateur saisi le code de déverrouillage correctement, le digicode

1. cf. `http://fr.wikipedia.org/wiki/Nombres_complexes`
2. cf. `http://fr.wikipedia.org/wiki/Nombre_rationnel`
3. cf. `http://fr.wikipedia.org/wiki/Digicode`

est déverrouillé.

Une interaction avec le digicode peut changer l'état de celui-ci (autrement dit, les objets instanciés sont modifiables). Puisque l'utilisateur interagit avec le digicode en appuyant sur une touche à la fois, il vous incombe d'ajouter d'autres attributs à votre classe (mais pas d'autres méthodes) pour assurer le bon fonctionnement de celle-ci.

Par souci de simplification, nous supposons que le fait d'appuyer sur un bouton lorsque le digicode est déverrouillé n'a aucun effet. Le digicode reste déverrouillé une fois qu'il l'est (heureusement que ce n'est pas le cas dans la vie réelle).

Version immuable

Créer une classe qui s'appelle `DigicodeI` qui reprend le même principe que la classe `DigicodeM` à l'exception qu'elle est immuable. Ainsi toute modification doit entraîner la création d'un nouvel objet et non pas la modification de l'état de l'objet existant.

Exercice 5 : Points 2D

Version immuable

Créer une classe appelée `Point2DI` qui permet de représenter des points à deux dimensions dans le plan euclidien. Cette classe doit permettre de créer des objets immuables et doit contenir les méthodes suivantes :

- Un constructeur par défaut `Point2DI()` qui permet de créer le point $(0, 0)$;
- Un constructeur `Point2DI(double x, double y)` qui permet de créer les points avec les abscisses et ordonnées passées en paramètre ;
- Les getters nécessaires pour observer l'état du point ;
- Une réécriture de la méthode `toString()` qui permet d'avoir un affichage des points sous la forme (x, y) ;
- Une méthode `distance(Point2DI p)` qui permet de calculer la distance entre le point appelant et celui passé en paramètre ;
- Une méthode `translate(double a, double b)` qui permet d'effectuer la translation du point avec le vecteur $\vec{u}\ (a, b)$;
- Une méthode `rotate(double theta)` qui permet d'effectuer la rotation avec un angle θ du point par rapport à l'origine.

Version modifiable

Coder maintenant la classe `Point2DM` équivalent à `Point2DI` mais qui permet d'avoir des points modifiables et non pas immuables. (Votre classe doit offrir la même panoplie de méthodes que la précédente).

Prenez le temps de tester vos classes en les instanciant dans des petits bouts de code de test afin de vous assurer de leur bon fonctionnement et afin de voir les différences qui existent entre manipulation d'objets modifiables et immuables.

Exercice 6 : Segments 2D

Un segment est une portion de droite délimitée par deux points A et B.

Version immuable

Coder la classe `Segment2DI` qui définit des segments de droites immuables dans le plan euclidien. Vous utiliserez les classes que vous avez définies dans l'exercice précédent pour représenter les points délimitant les segments. Votre classe doit contenir les méthodes suivantes :

- Un constructeur qui prend en paramètre les deux points délimitant le segment et qui crée ce dernier ;
- Les getters nécessaires pour observer l'état du segment ;
- La redéfinition de la méthode `toString()` afin d'avoir un affichage lisible du segments ;
- Une méthode `length()` qui calcule la longueur du segment ;
- Une méthode `translate(double a, double b)` qui permet d'effectuer la translation du segment avec le vecteur $\vec{u}\ (a, b)$;
- Une méthode `rotate(double theta)` qui permet d'effectuer la rotation avec un angle θ du segment par rapport à l'origine.

Version modifiable

Créer une deuxième classe `Segment2DM` qui est la version modifiable des segments 2D.

Exercice 7 : Réécriture de la méthode equals

Redéfinir, dans toutes les classes que vous venez de créer tout au long de ce TD, la méthode `equals(Object that)` - qui permet de comparer les objets - afin de fournir une égalité basée sur le contenu au lieu d'une égalité identitaire (ex. dans le cas des objets de type `Complexe`, la méthode retourne `true` si les deux objets ont la même partie réelle et la même partie imaginaire). Vous devez faire bien attention aux points suivants :

- Le cas où l'objet `that` passé en paramètre est le même que l'objet appelant.
- Le cas où l'objet `that` est égal à `null`.
- Dans les cas autres que les deux cas susmentionnés, vous devez vous assurer que l'objet `that` est une instance de la même classe que votre objet appelant (vous pouvez utiliser l'opérateur `instanceof`), dans ce cas, il faut "caster" `that` dans cette classe pour pouvoir accéder à ses attributs convenablement.

La signature de la méthode `equals` est la suivante : `public boolean equals(Object that)`. Vous devez impérativement adhérer à cette signature. Notamment, l'objet passé en paramètre doit obligatoirement avoir `Object` comme type.