Exercice 1. Héritage et polymorphisme

Le choix d'une méthode redéfinie ne dépend pas du type statique (fixé à la compilation), mais du type effectif de l'objet appelant la méthode (à l'exécution).

- Ecrire une classe `Voyage`. Un voyage est caractérisé par une ville de départ, une ville d'arrivée, une longueur (un `ReelContraint` *compris entre 1 et 30000 km ; cf. classe* `EntierContraint` *vue en cours*) ainsi que son prix au km (un `ReelContraint` *compris entre 0.05 et 3.2 euros*). Le prix d'un voyage est obtenu en multipliant sa longueur par son prix au km.
- Proposer une classe `VoyageReduit` qui hérite de la classe `Voyage` et possède un attribut `tauxPromotion` (un `ReelContraint` prenant une valeur entre 0.01 et 0.99).
- Redéfinir la méthode `getPrix` de la classe `Voyage`. On fera attention à réutiliser au maximum le code déjà écrit dans la classe mère. Il faut commencer par écrire cette méthode dans la classe `Voyage` si cela n'est pas encore fait.
- Tester la méthode redéfinie dans le cadre du polymorphisme, c'est-à-dire en stockant les références des types dérivés dans des références vers le type de base.
  - On pourra stocker des `Voyage` et des `VoyageReduit` dans un tableau de références sur des `Voyage` puis répondre à une requête utilisateur : est-ce que son voyage est disponible et si oui lequel est le moins cher (et le moins long si plusieurs voyages ont le prix le moins élevé).

Diagramme de classes proposé : http://yuml.me/edit/8f7a254e

Exercice 2. Héritage et la classe Object

- Reprendre la classe `Point2D` du TD3 (avec des coordonnées flottantes et un constructeur) et redéfinir la méthode `boolean equals(Object o)`. Vérifier dans vos tests que cette redéfinition suit bien les contraintes vues en cours.
- Faire implanter l'interface `Cloneable` à la classe `Point2D`, puis redéfinir la méthode `Object clone()`. Tester la méthode `clone`, en ne manipulant que des références sur des `Object` (le type effectif étant bien entendu `Point2D`).

Exercice 3. Héritage et classe abstraite

- Modéliser une classe abstraite `FigureGeometrique` qui impose au minimum les méthodes abstraites suivantes à ses classes dérivées :
  - `void homothetie(double coef)`
  - `void rotation(double angle)`
  - `void identite()`
- La classe abstraite `FigureGeometrique` sera pourvue d'une méthode d'instance `affiche` :
  - ```
    public void affiche()
    {
        Systeme.out.println("Je suis une figure geometrique de type :") ;
        identite() ;
    }
    ```
- Compléter votre classe abstraite avec les méthodes statiques (de classe) suivantes :
  - Une méthode `afficheFigures` pour afficher les figures géométriques contenues dans un tableau de `FigureGeometrique`.
    ```

    - Une méthode `homothetieFigures` pour appliquer une homothétie à toutes les figures d'un tableau de `FigureGeometrique` ; une méthode `rotationFigures` pour appliquer une rotation à toutes les figures.
- Est-ce qu'une méthode abstraite peut être déclarée privée ?

<u>Exercice 4.</u> Héritage et interface

```
public interface Affichable
{
   void affiche() ;
}
```

Voici une interface commune à toute forme géométrique 2D :

```
public interface FormeGeometrique2D extends Affichable
{
   void deplace(float dx, float dy) ;
   boolean estIdentique(Object o) ;
}
```

- Pourquoi n'est-il pas nécessaire de préciser "abstract public" avant chaque méthode d'une interface ?
- Reprendre l'exercice 3 du TD3 et faire implanter les interfaces `FormeGeometrique2D` et `Cloneable` à 2 classes parmi celles représentant une forme géométriques 2D (`Point2D`, `Segment2D`, `Triangle2D` et `Rectangle2D`).
    - Quels sont les possibilités offertes par cette démarche ?
- Lister les différences entre les classes abstraites et les interfaces.
- Quand est-il plus judicieux d'utiliser une interface qu'une classe abstraite ?

<u>Exercice 5.</u> Héritage et modélisation d'un problème

- Modéliser, grâce à l'héritage, les types concrets (donc instanciables) `Etudiant`, *`EtudiantBoursier`*, `PersonnelAdministratif`, `PersonnelEnseignantTitulaire` et `PersonnelEnseignantVacataire`. Vous pouvez introduire des classes intermédiaires afin de maximiser la réutilisation du code (et en particulier des classes abstraites ou des interfaces !). Vous devez fournir tous les services (méthodes publiques) que vous jugerez nécessaires (`getNom`, `getNumeroBureau`, `getSalaire`, `getMontantBourse` etc.).
- Tester toutes vos classes ainsi que leurs services.
- Déclarer `final` les classes qui doivent l'être. Est-ce que certaines méthodes doivent être déclarées `final` ?

<u>Exercice 6.</u> Les classes enveloppes

Tester les classes enveloppes vues en cours, et en particulier l'addition entre des types scalaires et des types enveloppes.