TD de Compléments en Programmation Orientée Objet n° 2 : Révisions : polymorphisme

Exercice 1 :

On suppose déjà définies :

```
1 | class A {} class B {} interface I {} interface J {}
```

Voici une liste de déclarations :

```
  | class C extends I {}
  | interface K extends B {}
  | class C implements J {}
4 | interface K implements B {}
  | class C extends A implements I {}
  | interface K extends I, J {}
  | class C extends A, B {}
```

Lesquelles sont correctes ?

Exercice 2 : Ordres

On se donne l'interface Ordre suivante :

```
interface Ordre {
  boolean plusGrandQue(Object other);
}
```

Écrivez et testez les classes suivantes qui implémentent l'interface Ordre :

- Reel : où plusGrandQue() correspond à l'ordre usuel sur les nombres réels.
- Diviseur où x.plusGrandQue(y) répond **true** si et seulement si l'entier y divise l'entier x.
- Prefixe où x.plusGrandQue(y) répond **true** si la chaîne de caractères y est préfixe de la chaîne x (on pourra utiliser x.startsWith(y)).

Remarque : si vous vous rappelez comment on fait, vous pouvez écrire une version générique, plutôt que convertir des Object en Integer ou Double.

Exercice 3 : Listes chaînées

On explore une façon particulière de programmer des listes chaînées pouvant contenir plusieurs types de données, mais sans utiliser la généricité.

Une liste chaînée est constituée de cellules à deux champs : un champ "contenu" (contenant un des *éléments* de la liste) et un champ "suivant", pointant sur une autre cellule, ou bien sur rien (fin de liste).

Pour notre mise en œuvre en Java, on va considérer que tout objet implémentant l'interface `Chainable`, ci-dessous peut servir de cellule de liste chaînée :

```
interface Chainable {
    Chainable suivant();
}
```

Ainsi, un objet Chainable peut représenter une liste non-vide (l'objet est la première cellule, les suivantes sont obtenues par appels successifs à la méthode suivant()), alors que la liste vide est juste représentée par la valeur **null**.

1. Écrivez les classes `EntierChainable` et `MotChainable` implémentant l'inferface `Chainable` et dont les objets contiennent respectivement un élément entier et un élément chaîne de caractères.
2. Écrivez pour chacune de ces classes le constructeur de types respectifs `public EntierChainable(int elt, Chainable suiv)`, et `public MotChainable(String elt, Chainable suiv)`, construisant une nouvelle cellule de contenu `cont` et de successeur `suiv`.
3. Programmez une méthode **int** longueur() donne la longueur d'une liste. Faites en sorte qu'il n'y ait pas besoin d'ajouter du code dans toutes les implémentations de Chainable.
4. Écrivez les méthodes `toString()` de ces classes. Elles devront non seulement présenter la donnée stockée dans la cellule, mais aussi celles des cellules suivantes.
5. Pourrait-on programmer la méthode toString() de la même façon que la méthode longueur () ? Que faudrait-il changer/ajouter à l'interface Chainable ?

On considère maintenant l'interface `Pesable` vue en cours :

```
interface Pesable {
    int poids();
}
```

4. Écrivez les classes `EntierPesableChainable` et `MotPesableChainable`, implémentant à la fois l'inferface `Chainable` et l'interface `Pesable`. Complétez les classes de l'exercice précédent en leur ajoutant leurs méthodes `poids()`. On considère que le poids d'un entier est sa valeur absolue, le poids d'une chaîne sa longueur, et le poids d'une liste sa somme des poids de ses cellules.
5. Pourrait-on programmer la méthode poids() de la même façon que la méthode longueur() ? Que faudrait-il changer/ajouter à l'interface Pesable ?

Exercice 4 : Personnes

```
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;
 Prof(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"
      );
 }
}

class Test {
 public static void main (String[] args){
  Personne jacques = new Personne ("
      Dupont");
  Enseignant victor = new Enseignant("
      Marsault","cpo");
  Personne aldric = new Enseignant("
      Degorre","cpo");
  victor.chante();
```

```
    jacques.enseigne();
    aldric.chante();
    aldric.enseigne();

    Personne[] comite = {jacques, victor,
        aldric};

    for(Personne p:comite)
      p.presenteToi();
  }
}
```

1. Lire attentivement le code, tout comprendre. Quand une instruction est fausse, corrigez-la ou supprimez-la.
2. Dans la classe Enseignant :
   — quels sont les attributs hérités, ajoutés ?
   — quels sont les méthodes héritées, ajoutées, redéfinies ?
   — comment fonctionne le constructeur ?
3. Qu'affiche le code ?
4. Qu'est-ce qui est polymorphe dans le code ?

Exercice 5 : Liaison dynamique

Qu'affiche le programme suivant :

```
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"); }
}

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
        }
}
```

Exercice 6 : Modélisation

Voici la situation :

— Un véhicule a un poids, un propriétaire, un conducteur, un numéro d'immatriculation.
— Certains objets peuvent être remorqués. D'autres objets peuvent tirer des objets remorquables. Chaque objet remorquable à son poids propre, mais peut aussi contenir une charge utile qui s'y additionne.
— Une fourgonette est un véhicule de transport d'objets dont le poids total est la somme de son poids propre et de sa charge.
— Les camions semi-remorque sont des véhicules capables de tirer une ou plusieurs remorques (pour peu que la première remorque tirée soit elle-même capable d'en remorquer une autre...). Le poids d'un semi-remorque est son poids propre, plus le poids de tout ce qu'il tire.
— Les transports de passagers sont des véhicules transportant un certain nombre de personnes. Le poids d'un transport de passagers est fonction du nombre de personnes transportées (70kg de plus par passager). Une voiture personnelle est un transport de passagers à 4 passagers maximum. Un autocar est un transort à 50 passagers maximum.

On voudrait modéliser cette situation en Java afin de répondre à différentes questions, en particulier liées à la sécurité des infrastructures routières (poids total des véhicules sur un pont, nombre d'humains au total dans un tunnel, etc.).

Proposez un ensemble de classes et d'interfaces modélisant cette situation :

1. Écrivez d'abord leur diagramme (dans un style à la UML : un rectangle par type, reliés par des flèches différentes en fonction qu'il s'agit d'une relation d'héritage ou d'implémentation. Chaque rectangle donne le nom de la classe ou interface, la liste de ses attributs, la liste de ses méthodes, en précisant à chaque fois **static** ou pas, **public** ou **private**, etc.).
2. Écrivez ensuite toutes ces classes et interfaces en Java (**class** A **extends** B **implements** I, J { attributs; methodes(); ... }).
3. Écrivez (dans une classe utilitaire auxiliaire) une méthode statique prenant en paramètre une liste (ou tableau) de véhicules et donnant le poids total des véhicules de la liste.
4. Écrivez une méthode prenant en paramètre une liste (ou tableau) de véhicules et donnant le nombre d'humains dans ces véhicules.
5. Écrivez une méthode prenant en paramètre une liste (ou tableau) de véhicules et donnant le poids total des objets transportés par ces véhicules et leurs remorques.