Université Paris Diderot **Programmation Orientée Objet**
L2-L3 **Année 2014-2015**

TD n° 7

Divers

Classes communes aux trois exercices suivants On considère la classe suivante :

```
public class MaDate{
    private int annee;
    private int mois;
    private int jour;

    public MaDate( int annee, int mois, int jour){
        if (mois <= 0 || mois >= 12)
            throw new IllegalArgumentException("mois doit etre compris entre 1 et 12");
        if (jour <= 0 || jour >= 31)
            throw new IllegalArgumentException("jour doit etre compris entre 1 et 31");

        // on devrait faire un test plus fin, par exemple, si mois =2 ,
        //jour doit etre <=28 si annee non bissextile, etc.

        this.annee = annee;
        this.jour = jour;
        this.mois = mois;
    }
}
```

On considère, par ailleurs, une classe `Personne` qui représente une personne avec son nom, son prénom, sa date de naissance et son adresse.

Exercice 1 (Méthode equals) Redéfinissez la méthode `equals(Object o)` dans la classe `Personne`. Si nécessaire vous pouvez ajouter des méthodes dans `MaDate`.

Exercice 2 (Interfaces Comparable et Comparator) On peut trier des listes chaînées (`LinkedList`) d'objets de type `T` grâce à la méthode statique `Collections.sort`. Bien entendu, pour qu'un tri soit possible, il faut que l'on indique comment on compare les éléments de type `T`. Il y a deux façons de faire cela :

La première solution est que le type `T` implémente l'interface `Comparable<T>`. (La méthode `sort` prend alors comme unique argument la liste chaînée.)

```
public interface Comparable<T>{
    int compareTo(T o);
}
```

On considère que la méthode `compareTo` correspond à "l'ordre naturel", c'est-à-dire celui qui est le plus "usuel".

L'appel `x.compareTo(y)` retournera un nombre strictement négatif si `x` est strictement inférieur à `y`, un nombre strictement positif si `x` est strictement supérieur à `y`, et 0 en cas d'égalité. Par ailleurs, il est fortement recommandé que cet ordre soit compatible avec la méthode equals().

La deuxième solution Consiste à fournir un ordre sous la forme d'un objet d'une classe implémentant l'interface `Comparator<T>`.

```
public interface Comparator<T>{
int compare(T o1, T o2)
     ///Compares its two arguments for order.
boolean equals(Object obj)
     //Indicates whether some other object is "equal to" this comparator.
}
```

La sémantique de la méthode `compare` est la même que celle de la méthode `compareTo`, à ceci près qu'on ne parle plus "d'ordre naturel". La méthode `sort` prend alors deux arguments : la liste chaînée de `T` et un objet de type `Comparator<T>`.

Questions (Comme dans l'exercice précédent, il peut être utile d'ajouter des choses dans la classe `Madate`.)

1. on considère que "l'odre naturel" entre deux personnes, consiste à regarder d'abord le nom, puis le prénom, puis la date de naissance, et, enfin, l'adresse. Par exemple, "Durand Paul né le 24 avril 1995" est "plus petit" que "Durand Paul né le 10 mars 2001".

   Faites en sorte que `Personne` implémente `Comparable<Personne>`.
2. On voudrait maintenant pouvoir trier un groupe de personnes en prenant comme premier critère la date d'anniversaire dans l'année. Par exemple, dans la liste triée "Dupuis Samia née le 24 avril 1995" doit être avant "Boulanger Akiko née le 10 novembre 1978". Comme on ne veut pas changer la précédente implémentation de Comparable dans la classe `Personne`, implémenter dans `Personne` une méthode qui renvoie un objet de type `Comparator` approprié.
3. À votre avis, pourquoi la méthode `compare` n'est pas statique ? Écrivez quelques lignes qui permettent de comparer deux objets de type `Personne` à l'aide de la méthode faite dans l'exo précédent.

Exercice 3 (Interface, classe abstraite) On considère maintenant une interface `Groupe` qui représente un groupe ordonné de personnes.

```
interface InterfaceGroupe{
   int taille(); // la taille du groupe

   // la nieme personne du groupe, la premiere personne a le numero 1
   Personne niemePersonne(int n);

   void ajouterEnFin(Personne p);

   // insere la personne p a la ieme position
   void ajouter(Personne p, int i);

   // supprime la première personne egale à p, retourne true si trouve
```

```
    boolean supprimer(Personne p);

    // supprime la ieme personne
    void supprimer(int i);

    // retourne un tableau contenant les personnes du groupe dans l'ordre
    // une suppression dans le tableau résultat ne modifiera pas this.
    Groupe[] versTab();
}
```

1. On veut écrire une classe abstraite `GroupeAbstrait` qui facilitera l'implémentation de l'interface, dans le sens où les classes concrètes qui en hériteront auront le moins de méthodes possibles à implémenter. Quelles méthodes peut-on rendre concrète dans cette classe ? (Il n'y a pas forcément de solution unique).
   Écrivez le code de cette classe.
2. Écrivez une classe concrète `GroupeListe` qui hérite de `GroupeAbstrait` en utilisant une liste chaînée pour stocker les personnes. Donnez un code de `Groupe` qui demande le moins de travail possible au programmeur.
3. Si on veut maintenant améliorer l'efficacité des méthodes, comment écririez-vous le code ?
4. Quelles autres méthodes pourriez-vous ajouter à l'interface `InterfaceGroupe` ?
5. Proposez d'autres manières d'implémenter l'interface `InterfaceGroupe`.