Université du Maine **MAN-M1** **Programmation Orienté Objet.**

TP n° 2 : Médiathèque et Rappels de cours

Question 1 : Ecrire une classe `Date` définie par le jour, le mois et l'année. On voudra pouvoir comparer deux dates en utilisant l'ordre naturel (cf. Annexe 1). Lors de la création d'une date, on vérifiera que le jour n'est pas supérieur à 31 et que le mois n'est pas supérieur à 12. Le cas échéant une `InvalidDateException` sera levée.

Question 2 : Ecrire la classe `Auteur` qui permet de représenter les auteurs des documents. Un auteur est défini par son nom, et ses dates de naissance et de décès. La date de décès est fixée à `null` si l'auteur est toujours vivant. On veut pouvoir tester si deux auteurs sont égaux (mêmes nom et dates de naissances/décès).

Question 3 : Ecrire la classe `Document` définie par un auteur et un titre. Pour cette classe, on redéfinira les méthodes `toString` et `equals`. De plus un document sait s'il est actuellement emprunté ou non. On prévoira une méthode pour modifier le statut emprunté/disponible d'un document.

Question 4 : (cf. Annexe 2) On considère le squelette de code suivant de la classe `BaseDocuments` qui permet de gérer la base documentaire de la médiathèque. En vous basant sur le code javadoc fourni, donnez un code java complet pour la classe `BaseDocuments` (et donc complétez aux endroits mentionnés).

```
public class BaseDocuments
{
COMPLETER // attribut(s) et constructeur(s)

/** ajoute un document dans la base de documents
* @param d le document `a ajouter
*/
COMPLETER méthode ajoute

/** supprime un document de la base de documents (si il est plusieurs
* fois présents, une seule des occurrences est supprimée). Il
* importe peu que le document soit emprunté ou non ,au moment de sa
* suppression.
* @param d le document à supprimer
* @return le document supprimé, null s'il n'existe pas
*/
COMPLETER méthode supprime

/** affiche tous les documents de la mediathèque (pour impression
* du catalogue par exemple)
*/
COMPLETER méthode affiche

/** indique si le document donné est disponible (non emprunté)
* @param d le document concerné
* @return true si le document donné est disponible (non emprunté)
* @exception NoSuchElementException si le document n'existe
* pas dans la mediathèque
*/
COMPLETER methode estDisponible

/** permet d'emprunter un document
* @param d le document concerne
* @exception NoSuchElementException si le document n'existe
* pas dans la mediathèque
*/
COMPLETER méthode emprunte

/** permet de rendre un document
* @param d le document concerné
*/
COMPLETER méthode rend
}
```

Question 5 : On souhaite à présent réaliser un moteur de recherche permettant de sélectionner un ensemble de documents de la base de documents selon des critères donnés.
Pour cela on définit une interface `Selectionneur` qui permet de déterminer les documents qui satisfont un critère. Cette interface contient une seule méthode : `public boolean estSatisfaitPar(Document d)` `s.estSatifaitPar(d)` est vrai quand `s` est satisfait par le document `d`.

Question 6 : Ajouter, à la classe BaseDocuments la méthode :
`public Iterator selectionne(Selectionneur s)` qui retourne un itérateur sur la collection des documents de la base de documents qui satisfont le sélectionneur `s`.

Question 7 : Créer un sélectionneur satisfait par les documents non empruntés, un autre satisfait par les documents dont l'auteur était encore vivant à une année donnée et un dernier satisfait par les documents dont le titre contient un mot m donné (on pourra utiliser la méthode `indexOf` de la classe `String`). Tester.

Question 8 : On veut pouvoir effectuer des sélections multi-critères, c'est-à-dire récupérer les documents qui satisfont simultanément plusieurs sélectionneurs. On parlera de sélectionneur composite. Un sélectionneur composite **est de type `Selectionneur`**. Ecrire la classe `SelectionneurComposite` qui réalise le « et logique » de plusieurs autres objets `Selectionneur`. Un `SelectionneurComposite` est un sélectionneur qui est composé se sélectionneurs (stockés dans une liste) et qui est satisfait par un document si tous les sélectionneurs de la liste. On prévoira une méthode **`add`** pour ajouter des sélectionneurs à la liste de sélectionneurs.

Question 9 : Afin de tester vos classes, vous écrirez le code nécessaire pour sélectionner les documents dont le titre contient le mot polymorphisme et dont l'auteur était vivant en 2005.

Question 10 : Créer une classe `TitreComparator` qui permet de comparer deux documents selon leur titre (par ordre alphabétique).

Question 11 : Ajouter, à la classe `BaseDocuments`, la méthode de classe `listeTriee(Iterator i, Comparator c)` qui retourne une liste triée avec le comparateur `c`, des éléments de l'itérateur `i`.

Question 12 : Ecrire le code nécessaire pour obtenir une liste triée selon le titre de tous les documents dont le titre contient polymorphisme.

Annexe 1 : Ordre sur les Objets

Il y a deux manières de définir un ordre sur des objets,

- soit de façon externe, où on introduit un objet comparateur qui est capable de dire si deux objets sont dans l'ordre ;
- soit de façon interne où ce sont les objets eux-mêmes qui sont capables de se comparer entre-eux, on parle alors **d'ordre naturel**.

La première manière est la plus souple puisqu'elle permet de définir plusieurs ordres différents pour un même type d'objet. Ceci est indispensable si on veut par exemple pouvoir parcourir une promotion d'étudiants par noms croissants et, séparément, par moyennes décroissantes.

A- Ordre défini par un Comparator

Il s'agit de donner une implémentation de l'interface java.util.Comparator :

```
public interface Comparator {
  public int compare(Object o1, Object o2) ;
  public boolean equals(Object obj) ;
}
```

La méthode principale est `compare()` qui renvoie un entier :

- négatif, si `o1` est strictement inférieur à `o2`
- nul, si `o1` est égal à `o2`
- Positif, si `o1` est strictement supérieur à `o2`

Il faut bien sûr que `o1` et `o2` soient comparables, c'est à dire qu'ils doivent être compatibles avec un type commun qui permet de définir la relation d'ordre recherchée. Si ce n'est pas le cas, il suffit de déclencher une exception `ClassCastException`. C'est très simple puisque cette exception est automatiquement déclenchée par une indication de type invalide.
Cette technique permet d'ordonner la collection d'objets suivant plusieurs ordres : il suffit d'implémenter autant de fois que nécessaire cette interface.

Par exemple, on veut disposer de deux ordres différents sur des étudiants : un ordre par noms croissants et un ordre par moyennes décroissantes. Ici le type commun avec lequel doivent être compatibles les deux objets est tout simplement `Etudiant`.

```
class NomsCroissants implements java.util.Comparator {
  public int compare (Object o1, Object o2) {
    return compare ((Etudiant) o1, (Etudiant) o2) ;
  }
  private int compare (Etudiant e1, Etudiant e2) {
    return e1.nom ().compareTo (e2.nom ()) ;
    // Ordre natuel de String
  }
}

class MoyennesDecroissantes implements java.util.Comparator {
    public int compare (Object o1, Object o2) {
        return compare ((Etudiant) o1, (Etudiant) o2) ;
    }
    private int compare (Etudiant e1, Etudiant e2) {
        int me1 = e1.moyenne () ;
        int me2 = e2.moyenne () ;
        return me1 <  me2 ? 1 : me1 == me2 ? 0 : -1 ;
    }
}
```

B- Définir un ordre naturel

Tout objet qui implémente l'interface `java.lang.Comparable` est dit posséder un ordre naturel.

```
public interface Comparable {
     public int compareTo(Object o) ;
 }
```

La méthode `compareTo()` renvoie un entier :

- négatif, si `this` est strictement inférieur à `o`
- nul, si `this` est égal à `o`
- positif, si `this` est strictement supérieur à `o`

Cette solution est moins souple que la précédente car :

- Un objet possède au plus 1 ordre naturel,
- Pour qu'un objet ait un ordre naturel, il faut que sa classe implémente `Comparable` (autrement dit, il faut le prévoir à l'avance).

Par exemple on peut donner un ordre naturel aux `Etudiant` par une extension de `Etudiant` :

```
class EtudiantParNom extends Etudiant implements Comparable {
   public EtudiantParNom (String nom) { super (nom) ; }
   public int compareTo (Object o) {
     return this.nom ().compareTo (((EtudiantParNom) o).nom ()) ;
   }
}
```

Ici le type commun avec lequel doivent être compatibles les deux objets est tout simplement `Etudiant`.

Consistance avec equals()

La relation d'ordre doit toujours être consistante avec equals() :

Dans le cas d'un ordre naturel :

`e1.compareTo((Object)e2) == 0` renvoie la même valeur que `e1.equals((Object)e2)`

Dans le cas d'un ordre explicite :

`compare((Object)e1, (Object)e2)==0` renvoie la même valeur que `e1.equals((Object)e2)`

Si cette règle n'est pas respectée, il peut y avoir des comportements incohérents.

Annexe 2 : Les Conteneurs Java

Lors de leur exécution, les programmes que vous écrirez créeront souvent de nouveaux objets basés sur des informations qui ne seront pas connues avant le lancement du programme.
Le nombre voire même le type des objets nécessaires ne seront pas connus avant la phase d'exécution du programme. Il faut donc être capable de créer un certain nombre d'objets, de les stocker et de les manipuler. Le simple usage de références nommées sur ces objets ne peut suffire.
Java propose une bibliothèque de classes qui fournit des implémentations de nombreuses structures de données permettant de stocker et de manipuler des références d'objets.

La bibliothèque de conteneurs de Java 2 se divise en deux concepts distincts :

- `Collection` : un groupe d'éléments individuels, souvent associés à une règle définissant leur comportement, par exemple :
  - dans une `List` les éléments sont ordonnés ;
  - dans un `Set` il ne peut y avoir de doublon.
- `Map` : un ensemble de paires clef – valeur.
  - Il est possible d'extraire un sous-ensemble d'une `Map` dans une Collection.
  - Une `Map` peut renvoyer un `Set` de ses clefs, une `Collection` de ses valeurs, etc.

Le type des objets

Les conteneurs sont conçus pour stocker n'importe quel type d'objet. Les conteneurs stockent donc des références sur des `Object` puisque quelque soit son type, un objet descend toujours de la classe `Object`. Ceci a pour but la généricité des conteneurs, et permet de stocker des objets hétérogènes dans un même conteneur. Cette solution a toutefois 2 inconvénients :

- Puisque l'information de type est ignorée lorsqu'on stocke une référence dans un conteneur, on ne peut placer aucune restriction sur le type de l'objet stocké dans le conteneur, même si on l'a créé pour ne contenir, par exemple, que des chats. Quelqu'un pourrait très bien ajouter un chien dans le conteneur.
- Puisque l'information de type est perdue, la seule chose que le conteneur sache est qu'il contient une référence sur un objet. Il faut réaliser un transtypage sur le type adéquat avant de l'utiliser.

```
import java.util.*;

public class CatsAndDogs {
  public static void main(String[] args) {
    ArrayList cats = new ArrayList();
    for(int i = 0; i < 7; i++)
      cats.add(new Cat(i));
    // Ce n'est pas un problème
    // d'ajouter un chien parmi les
    // chats :
    cats.add(new Dog(7));
    for(int i = 0; i < cats.size(); i++)
      ((Cat)cats.get(i)).print();
    // Le chien est détecté seulement
    // lors de l'exécution.
  }
}
```

```
public class Cat {
  private int catNumber;
  Cat(int i) { catNumber = i; }
  void print() {
    System.out.println("Cat #" + catNumber);
  }
}
```

```
public class Dog {
  private int dogNumber;
  Dog(int i) { dogNumber = i; }
  void print() {
    System.out.println("Dog #" + dogNumber);
  }
}
```

Créer une ArrayList consciente du type

```
public class MouseList {
  private List list = new ArrayList();
  public void add(Mouse m) {
    list.add(m);
  }
  public Mouse get(int index) {
    return (Mouse)list.get(index);
  }
  public int size() {
    return list.size(); }
}
```

```
public class MouseListTest {
 public static void main(String[] args){
    MouseList mice = new MouseList();
    for(int i = 0; i < 3; i++)
        mice.add(new Mouse(i));
    for(int i = 0; i < mice.size(); i++)
        MouseTrap.caughtYa(mice.get(i));
  }
}
```

Pourquoi `MouseList` contient une `ArrayList` et ne dérive pas de `ArrayList` ?

Si `MouseList` avait été dérivée de `ArrayList`, la méthode `add(Mouse)` aurait simplement surchargé la méthode existante `add(Object)` et aucune restriction sur le type d'objets acceptés n'aurait donc été ajoutée.

Afficher les conteneurs

Pour afficher un conteneur on peut simplement utiliser la méthode :
`System.out.println`

- Affichage d'une `ArrayList` :
  `[dog, dog, cat]`
- Affichage d'un `HashSet` :
  `[cat, dog]`
- Affichage d'une `HashMap` :
  `[cat=Rags, dog=Spot]`

Les Itérateurs

Un itérateur est un objet dont le travail est de se déplacer dans une séquence d'objets et de sélectionner chaque objet de cette séquence sans que le programmeur client n'ait à se soucier de la structure sous-jacente de cette séquence.
Un itérateur est généralement ce qu'il est convenu d'appeler un objet « léger » : un objet bon marché à construire. Pour cette raison, vous trouverez souvent des contraintes étranges sur les itérateurs ; par exemple, certains itérateurs ne peuvent se déplacer que dans un sens.

Avec un `Iterator` Java On ne peut pas faire grand-chose mis à part :

- Demander à un conteneur de renvoyer un `Iterator` en utilisant une méthode appelée `iterator()`. Cet `Iterator` sera prêt à renvoyer le premier élément dans la séquence au premier appel à sa méthode `next()`.
- Récupérer l'objet suivant dans la séquence grâce à sa méthode `next()`.
- Vérifier s'il reste encore d'autres objets dans la séquence via la méthode `hasNext()`.
- Enlever le dernier élément renvoyé par l'itérateur avec la méthode `remove()`.

```
import java.util.*;

public class CatsAndDogs2 {
 public static void main(String[] args){
 List cats = new ArrayList();
 for(int i = 0; i < 7; i++)
      cats.add(new Cat(i));
 Iterator e = cats.iterator();
 while(e.hasNext())
   ((Cat)e.next()).print();
 }
}
```

```
class Hamster {
  private int hamsterNumber;
  Hamster(int i) {
      hamsterNumber = i;
  }
  public String toString() {
     return "This is Hamster #" +
              hamsterNumber;
  }
}
```

```
class Printer {
  static void printAll(Iterator e) {
    while(e.hasNext())
    System.out.println(e.next());
  }
}
public class HamsterMaze {
  public static void main(String[] args) {
    List v = new ArrayList();
    for(int i = 0; i < 3; i++)
      v.add(new Hamster(i));
    Printer.printAll(v.iterator());
  }
}
```

Classification des conteneurs

Les interfaces concernées par le stockage des objets sont `Collection`, `List`, `Set` et `Map`.
Idéalement, la majorité du code qu'on écrit sera destinée à ces interfaces, et le seul endroit où on spécifiera le type précis utilisé est lors de la création.

L'interface List

Les `Lists` sont déclinées en deux versions :

- l'`ArrayList` de base, qui excelle dans les accès aléatoires aux éléments,
- la `LinkedList`, bien plus puissante (qui n'a pas été conçue pour un accès aléatoire optimisé, mais dispose d'un ensemble de méthodes bien plus conséquent).

On pourra donc créer une List de cette manière :

```
List x = new LinkedList();
```

Bien sûr, on peut aussi décider de faire de `x` (de l'exemple précédent) une `LinkedList` (au lieu d'une `List` générique) et véhiculer le type précis d'informations avec `x`. L'intérêt de l'utilisation d'une interface est que si on décide de changer l'implémentation, la seule chose qu'on ait besoin de changer est l'endroit où la liste est créée, comme ceci :

```
List x = new ArrayList();
```

Et on ne touche pas au reste du code

L'interface Set

Les `Sets` ont exactement la même interface que les `Collections`, et à l'inverse des deux différentes `Lists`, ils ne proposent aucune fonctionnalité supplémentaire.

Les `Sets` sont donc juste une `Collection` ayant un comportement particulier. Un `Set` refuse de contenir plus d'une instance de chaque objet. Les `Objects` ajoutés à un `Set` doivent définir la méthode `equals()` pour pouvoir établir l'unicité de l'objet. Un `Set` possède la même interface qu'une `Collection`. L'interface `Set` ne garantit pas que les éléments sont maintenus dans un ordre particulier.

- La classe `HashSet`
  - Pour les `Sets` où le temps d'accès aux éléments est primordial.
  - Les `Objects` doivent définir la méthode `hashCode()`.
- La classe `TreeSet`
  - Un `Set` trié stocké dans un arbre. De cette manière, on peut extraire une séquence triée à partir du `Set`.

L'interface Maps

Un tableau associatif, ou map, ou dictionnaire offre une fonctionnalité particulièrement puissante de « sélection dans une séquence ». Conceptuellement, cela ressemble à une `ArrayList`, mais au lieu de sélectionner un objet par un indice, on le sélectionne en utilisant un autre objet !

- La méthode `put(Object key, Object value)` ajoute une valeur (la chose qu'on veut stocker), et l'associe à une clef (la chose grâce à laquelle on va retrouver la valeur).
- La méthode `get(Object key)` renvoie la valeur associée à la clef correspondante.
- Il est aussi possible de tester une `Map` pour voir si elle contient une certaine clef ou une certaine valeur avec les méthodes `containsKey()` et `containsValue()`.

La bibliothèque Java standard propose deux types de `Maps` : `HashMap` et `TreeMap`.

- Les deux implémentations ont la même interface (puisqu'elles implémentent toutes les deux `Map`), mais diffèrent sur un point particulier : les performances.
- Au lieu d'effectuer une recherche lente sur la clef, un `HashMap` utilise une valeur spéciale appelée code de hachage (hash code).
  - Le code de hachage est une façon d'extraire une partie de l'information de l'objet en question et de la convertir en un `int` « relativement unique ».
  - Tous les objets Java peuvent produire un code de hachage, et `hashCode()` est une méthode de la classe racine Object. Un `HashMap` récupère le `hashCode()` de l'objet et l'utilise pour retrouver rapidement la clef. Le résultat en est une augmentation drastique des performances.

À partir du JDK 1.2 on dispose de la possibilité de maîtriser l'ordre dans lequel se présenteront les éléments d'un `Set` ou d'une `Map` lors de l'utilisation d'un itérateur.

Ce sont les implémentations `TreeSet` de `SortedSet` et `TreeMap` de `SortedMap` qui fournissent cette fonctionnalité.

Conseils

- Utiliser une `ArrayList` si on doit effectuer un grand nombre d'accès aléatoires, et une `LinkedList` si on doit réaliser un grand nombre d'insertions et de suppressions au sein de la liste.
- Le comportement de files et piles est fourni via les `LinkedLists`.
- Une `Map` est une façon d'associer non pas des nombres, mais des objets à d'autres objets. La conception d'un `HashMap` est focalisée sur les temps d'accès, tandis qu'un `TreeMap` garde ses clefs ordonnées, et n'est donc pas aussi rapide qu'un `HashMap`.
- Un `Set` n'accepte qu'une instance de valeur de chaque objet. Les `HashSets` fournissent des temps d'accès optimaux, alors que les `TreeSets` gardent leurs éléments ordonnés.
- Il n'y a aucune raison d'utiliser les anciennes classes `Vector`, `Hashtable` et `Stack` dans du nouveau code.