Université Paris Diderot **Programmation Orientée Objet**
L2 **Année 2016-2017**

TD - Séance n°10

Génériques, interfaces Comparable et Comparator

Exercice 1 (Génériques) On souhaiterait créer une classe `Bazar<E>` ayant le fonctionnement suivant : c'est un ensemble d'objet de type `E`, ayant une taille fixée, auquel on peut ajouter/retirer des objets, compter les objets dedans, savoir s'il est vide. Seulement, lorsqu'on en retire un objet, celui-ci est pris aléatoirement dans les objets présents. On peut voir la classe `Bazar<E>` comme un mélangeur.

1. Commençons à écrire la classe `Bazar<E>` : dotez-la d'un constructeur avec un argument entier spécifiant sa taille.
2. Ecrivez maintenant ses méthodes `void inserer(E e)`, `E prendre()`, `int combienDedans()`, `boolean estVide()` et `boolean estPlein()`.
   Attention, en Java l'utilisation de tableau de génériques est complexe. Contentons-nous d'une `ArrayList` ou d'une `LinkedList` pour l'instant.
3. Faire que les méthodes `inserer` et `prendre` lèvent des exceptions de type `BazarPleinException` et `BazarVideException`, qui étendent toutes les deux `RuntimeException`. Quelle différence aurait-on si ces exceptions étendaient `Exception` à la place ?
   Un exemple d'utilisation de `Bazar<E>` est le suivant :

```
public int[] loto{ // renvoie 6 numéros entre 1 et 49,
                                          sans répétition
    Bazar<Integer> urne = new Bazar<Integer>(49);
    for(int i = 1; i < 49; i++)
        urne.inserer(i);

    int[] resultat = new int[6];
    for(int i = 0; i < 6; i++)
        resultat[i] = urne.prendre(); //tirage des numéros

    return resultat;
}
```

4. On aimerait maintenant pouvoir utiliser le même objet `Bazar<Integer>` pour un deuxième tirage, indépendant et différent du premier. Ajouter une méthode `void vider()` qui vide un `Bazar<E>` de tous les `E` qu'il contient, de façon à pouvoir réinitialiser une urne entre deux tirages.

Exercice 2 (Interfaces Comparable et Comparator) Les classes implémentant l'interface `Collection<T>` (attention, sans s) comme les listes chaînées (`LinkedList<T>`), les tableaux dynamiques(`ArrayList<T>`) et bien d'autres, peuvent être triées par la méthode statique `Collections.sort` (attention, avec un s). Un tableau peut être trié de la même manière par la méthode `Arrays.sort`.

Un tri nécessite néanmoins qu'on puisse comparer deux éléments entre eux ! Pour cela, deux solutions :

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

```
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 signature de `Collections.sort` à un seul argument est : `static void sort(<T extends Comparable<? super T»)`

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>`.

Exercice 3 (Comparable, Comparator, et Enum : un jeu de carte)

On va créer des classes qui seraient une bonne base pour créer un jeu de carte ensuite.

1. Créer une classe `Carte`. On définira un `enum CouleurCarte` dont les valeurs sont `Pique, Coeur, Carreau, Trefle`.
2. Ajouter une méthode `equals(Carte c)` dans la classe `Carte`.

3. Sachant qu'un `Enum E` implémente automatiquement l'interface `Comparable<E>` (l'ordre est celui de la déclaration des `Enum`), et considérant que l'ordre naturel des cartes est d'abord selon la couleur, puis par valeur ascendante, faites que `Carte` implémente l'interface `Comparable<Carte>`. Pour les valeurs des cartes, on pourra se contenter d'un `int` entre 0 (as) et 12 (roi) (mais on pourrait également utiliser un `Enum` !).

   On aimerait maintenant créer une classe `OrdreCarteModifiable` implémentant l'interface `Comparator<Carte>`, dont on souhaiterait les fonctionnalités suivantes :

   (a) une méthode `void setAtout(CouleurCarte c)`, qui définit une couleur ayant précédence sur toutes les autres, ainsi qu'une méthode `removeAtout` qui enlève cet avantage d'une couleur sur les autres.

   (b) une méthode `void setCouleurCourante(CouleurCarte c)` qui indique la couleur jouée à un tour de jeu, et `void removeCouleurCourante` qui fait que le tour joué n'a pas de couleur courante.

   (c) une méthode `int meilleureCarte(Carte[] main)` qui renvoie l'indice de la meilleure carte d'un ensemble de cartes (la première meilleure carte en cas d'égalité).

   On pourrait ainsi utiliser ce comparateur pour décider du gagnant d'un tour de jeu en comparant les cartes entre elles.

4. Ecrire la classe `OrdreCarteModifiable`.

5. Imaginez comment on pourrait utiliser cet objet lors d'un tour de jeu, comment la modifier pour pouvoir représenter un maximum de jeux de cartes existant. (Idée : on pourrait vouvoir modifier l'ordre des valeurs des cartes, en général ou au sein d'une couleur...)

Exercice 4 (Génériques et Comparable) On va mettre ces idées en pratique avec notre classe `Bazar<E>`.

Dans certains cas, un `Bazar<E>` peut contenir des objets implémentant l'interface `Comparable<E>`. Dans ce cas, on souhaiterait pouvoir trier les éléments contenus puis pouvoir accéder au plus grand ou au plus petit. On va pour cela créer une classe dérivée `BazarTriable<E>` avec ces fonctionnalités.

1. Ecrire une classe `BazarTriable<E extends Comparable<E> > extends Bazar<E>` permettant de trier ses éléments, prendre le premier ou le dernier élément (méthodes `trier`, `prendrePremier`, `prendreDernier`).

2. Ecrire une classe `PaquetDeCarteStandard` qui étend `BazarTriable<Carte>` et possédant un constructeur sans argument qui crée un paquet contenant les 52 cartes habituelles.

3. A l'aide de la méthode `equals` de `Carte`, écrivez une méthode `void retirerDoublons()` pour votre `PaquetDeCarteStandard`.