Exercices INF3140 : série #6

Note : Voir plus loin (p. **??**) les explications concernant l'interface `Comparable` et la méthode `compareTo`.

1. Mise en oeuvre impérative de fonctions sur des ensembles

En utilisant les opérations de la bibliothèque `OclCollections`, écrivez du code Java réalisant les fonctions indiquées plus bas — un exemple d'utilisation est indiqué (cas de test JUnit). Votre mise en oeuvre doit utiliser une approche **impérative** standard — avec variables et boucles `for`.

a.
```
/** Determine l'element maximum parmi un ensemble d'entiers non negatifs.
 * Les elements n'etant jamais negatifs, il est donc possible et acceptable
 * de retourner 0 si l'ensemble est vide (de la meme facon que la somme
 * des elements d'un ensemble vide donne 0).
 *
 * @param ens l'ensemble a traiter
 * @return l'element maximum parmi les divers elements de l'ensemble ens
 */
static public Integer maximum( final Set<Integer> ens ) {
  ...
}


@Test public void exemple_maximum() {
   assertEquals( (Integer) 100,
                 (Integer) FonctionsEnsembles.maximum( mkSet(20, 100, 10, 20, 23, 34) ) );
}
```

b.
```
/** Determine l'element minimum parmi un ensemble d'elements (de
 * type arbitraire, mais qui peuvent etre compares entre eux).
 *
 * @param ens l'ensemble a traiter
 * @return l'element minimum parmi les divers elements de l'ensemble ens
 */
static public <T extends Comparable<T>> T minimum( final Set<T> ens ) {
  ...
}


@Test public void exemple_minimum() {
   assertEquals( (Integer) 10,
                 (Integer) FonctionsEnsembles.minimum( mkSet(20, 100, 10, 10, 20, 34) ) );
}
```

c.
```
/** Retourne une sequence, ordonnee (en ordre croissant),
 * contenant chacun des elements de l'ensemble de depart.
 *
 * @param ens l'ensemble a traiter
 * @return la sequence ordonnee avec tous les elements de l'ensemble ens
 */
static public <T extends Comparable<T>> Sequence<T> elementsOrdonnes( final Set<T> ens ) {
  ...
}
```

```
@Test public void exemple_elementsOrdonnes() {
   assertEquals( mkSequence(10, 20, 23, 34),
                 FonctionsEnsembles.elementsOrdonnes( mkSet(20, 10, 10, 20, 23, 34) ) );
}
```

d.

```
/** Etant donne un ensemble d'ensembles, retourne un ensemble qui
 * contient les elements qui sont communs a chacun des ensembles.
 *
 * @param ens l'ensemble a traiter
 * @return l'ensemble des elements communs a chacun des ensembles de l'ensemble ens
 */
static public <T> Set<T> elementsCommuns( final Set<Set<T>> ens ) {
  ...
}


@Test public void exemple_elementsCommuns() {
   assertEquals( mkSet( 20 ),
                 FonctionsEnsembles.elementsCommuns( mkSet(
                                                       mkSet(10, 20, 20),
                                                       mkSet(20, 20, 20),
                                                       mkSet(30, 20),
                                                       mkSet(20, 10, 10) ) ) );
}
```

e.

```
/** Etant donne un ensemble d'entiers arbitraires, retourne le
 * sous-ensemble des elements qui sont compris entre deux bornes.
 *
 * @param ens l'ensemble a traiter
 * @param inf la borne inferieure
 * @param sup  la borne superieure
 * @return l'ensemble des chaines pour les elements positifs de ens
 */
static public Set<Integer> elementsDansIntervalle( final Set<Integer> ens,
                                                   final Integer inf,
                                                   final Integer sup ) {
  ...
}


@Test public void exemple_elementsDansIntervalle() {
   assertEquals( mkSet(20, 23, 34),
                 FonctionsEnsembles.elementsDansIntervalle( mkSet(20, 100, 20, 10, 10, 20, 23, 34), 20
}
```

2. Mise en oeuvre «à la OCL» de fonctions sur des ensembles

En utilisant les opérations de la bibliothèque `OclCollections`, écrivez du code Java réalisant les fonctions indiquées plus bas — un exemple d'appel est présenté pour chaque fonction (à l'aide d'un cas de test JUnit).

Votre mise en oeuvre doit utiliser «le style OCL». En d'autres mots, vous devez éviter d'écrire du code Java impératif (avec variables et boucles `for`) ; vous devez plutôt utiliser des *expressions Java* qui ressemblent le plus possible à ce qu'on écrirait en OCL.

De plus, vous devez aussi spécifier, à l'aide d'une instruction `assert`, une pré-condition appropriée — utilisez «`assert true;`» si aucune condition particulière n'est nécessaire... mais n'oubliez pas qu'en Java, une référence reçue en argument peut être `null` :(

a.
```
static public Set<Integer> elementsDansIntervalle( final Set<Integer> ens,
                                                   final Integer inf,
                                                   final Integer sup ) {
  ...
}
```

b.
```
/** Etant donne un ensemble d'entiers arbitraires, retourne un
 * ensemble contenant la chaine (toString) pour les differents
 * entiers de l'ensemble de depart qui sont positifs.
 *
 * @param ens l'ensemble a traiter
 * @return l'ensemble des chaines pour les elements positifs de ens
 */
static public Set<String> stringDesPositifs( final Set<Integer> ens ) {
  ...
}

@Test public void exemple_stringDesPositifs() {
   assertEquals( mkSet( "10", "20", "23", "34" ),
                 FonctionsEnsembles.stringDesPositifs( mkSet(20, -100, 10, 10, -20, 23, 34) ) );
}
```

c.
```
static public Integer maximum( final Set<Integer> ens ) {
  ...
}
```

d.
```
static public <T extends Comparable<T>> T minimum( final Set<T> ens ) {
  ...
}
```

L'interface `Comparable` et les comparaisons d'objets

L'interface `Comparable` et la méthode `compareTo`

La bibliothèque standard Java définit l'interface générique suivante :

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

  // Parameters:
  //    o - the Object to be compared.
  // Returns:
  //   a negative integer, zero, or a positive integer as this object is
  //   less than, equal to, or greater than the specified object.
  // Throws:
  //   ClassCastException - if the specified object's type prevents it
  //   from being compared to this Object.
}
```

L'extension (sous-typage) du type `Comparable`

Une spécification d'un type `T` tel que `T extends Comparable<T>` signifie que `T` est un sous-type de `Comparable<T>`, en d'autres mots, que les valeurs de type `T` peuvent être comparées entre elles avec `compareTo` — le type `T` étant un sous-type de `Comparable<T>`, il supporte donc une opération `compareTo`.

Style suggéré pour les comparaisons

Pour `x` et `y` d'un type `T` tel que `T extends Comparable<T>`, on a alors les comparaisons et équivalences suivantes, qu'il est préférable d'utiliser, question de «style» (en d'autres mots, il est préférable d'éviter les comparaisons explicites du résultat de `compareTo` avec des valeurs spécifiques autres que 0) :

Java	Logique
`x.compareTo(y) < 0`	`x < y`
`x.compareTo(y) > 0`	`x > y`
`x.compareTo(y) <= 0`	`x <= y`
`x.compareTo(y) >= 0`	`x >= y`
`x.compareTo(y) == 0`	`x = y`

Signalons que la documentation indique ce qui suit quant à la relation entre `compareTo` et `equals` :

> *It is strongly recommended, but not strictly required that (x.compareTo(y)==0) == (x.equals(y)). Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."*