IN201 : 14 – Généricité

Author : Christophe Garion <garion@isae.fr>
Public : SUPAERO 2A
Date :

Solution

Résumé

Ce TP a pour but de vous faire créer et manipuler un type générique.

1 Contenu

Ce document est un corrigé succint du TP portant sur les observateurs. Les sources Java des classes sont disponibles sur le site http://www.tofgarion.net/lectures/IN201.

2 Problématique

On cherche ici à modéliser et à implanter un ensemble d'objets qui sont comparables entre eux (comme par exemple des entiers). Cet ensemble d'objets sera un type générique utilisant un paramètre de type formel. De plus, on souhaite pouvoir créer des ensembles d'ensembles comparables.

3 Conception de la classe `Ensemble`

Les ensembles que nous allons considérer seront des ensembles d'objets comparables entre eux. Les ensembles eux-mêmes seront également comparables entre eux. L'interface `Comparable` fournit un type représentant des objets comparables. Tous les objets de type `Comparable` possédent une méthode `compareTo` qui renvoie un entier. On consultera la documentation javadoc de l'interface `Comparable` pour plus de détails.
On souhaite pouvoir effectuer les opérations suivantes sur un ensemble :
– ajouter un élément à l'ensemble ;
– enlever un élement de l'ensemble ;
– obtenir un itérateur sur l'ensemble ;
– récupérer le minimum de l'ensemble ;
– construire l'union de deux ensembles de types compatibles.
Proposer un diagramme UML de la classe `Ensemble`.

Solution :
Une proposition de diagramme de classe est présentée sur la figure 1. Rien de bien particulier ici, l'utilisation du paramètre formel de type ne posait pas de problème. Attention, la notation `T[]` en UML pour l'attribut n'impose pas l'utilisation d'un tableau. Elle signifie juste que ens est un ensemble d'objets de type `T`. On remarquera que deux exceptions peuvent être levées lorsque l'on essaye de retirer un élément qui n'appartient pas à l'ensemble ou de calculer le minimum d'un ensemble vide. La méthode qui renvoie un itérateur s'appelle `iterator`, les raisons de ce choix sont expliqués dans la section 4.

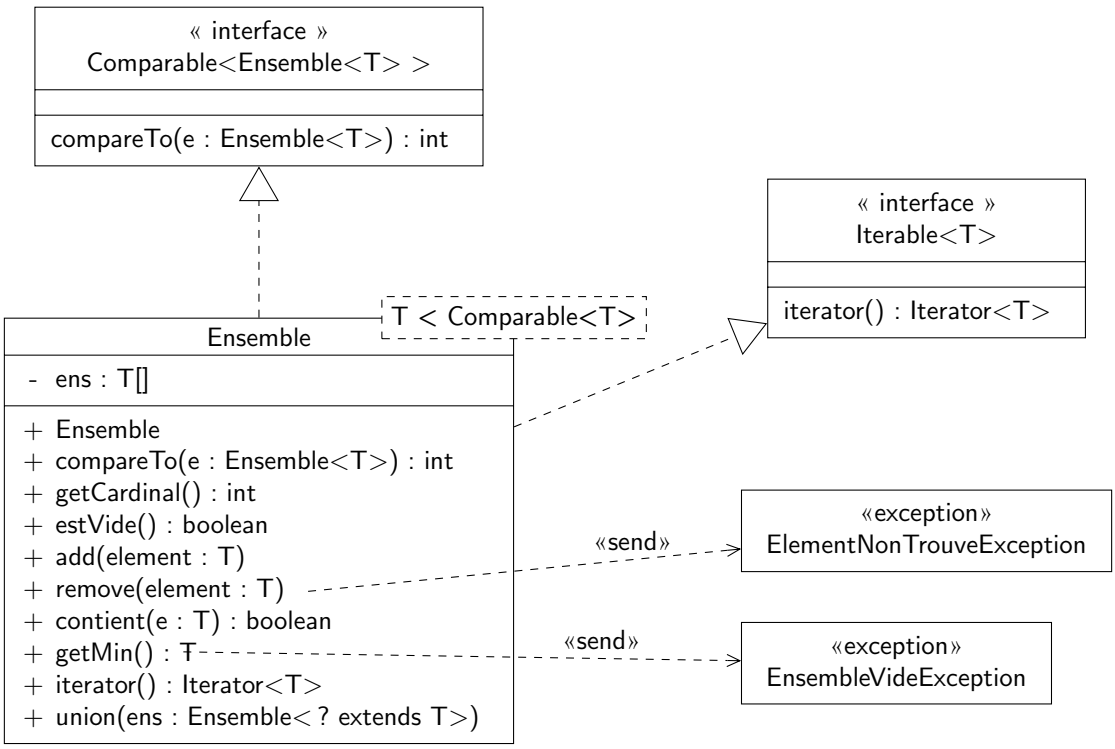

FIGURE 1 – Diagramme de la classe `Ensemble`

4 Implantation de la classe `Ensemble`

On développera la classe `Ensemble` en prenant soin de tester au fur et à mesure les méthodes que l'on développe.
On se demandera en particulier quel est le type de l'attribut de la classe `Ensemble` permettant de stocker les éléments (on pourra choisir une instance d'une collection).
On pourra également lever des exceptions si cela est nécessaire (minimum d'un ensemble vide etc).

Solution :
Vous trouverez les sources de la classe sur le site.
J'ai choisi d'utiliser une instance de `java.util.ArrayList` pour stocker les éléments de l'ensemble. Ceci permettait de récupérer facilement un itérateur sur l'ensemble en utilisant la méthode `iterator` de `ArrayList`. Toutes les collections de Java proposent d'ailleurs une méthode `iterator` qui renvoie un itérateur.
Je suis même allé plus loin. Si l'on jette un coup d'œil à la javadoc sur le site de Sun ou en local, on s'aperçoit que toutes les classes réalisant l'interface `Iterable<E>` peuvent être utilisées dans la boucle spéciale **for** que nous avons utilisée pour parcourir une instance de `Vector` par exemple. La seule méthode à implanter est cette fameuse méthode `iterator`. C'est pour cela que la méthode de `Ensemble` s'appelle `iterator`. J'en profite pour faire réaliser `Iterable<T>` par `Ensemble` et on peut alors utiliser la boucle **for** pour un ensemble (cf. programme de test développé dans la section 5).
La méthode `toString` de la classe est assez « évoluée » : dans un premier temps, si l'ensemble est vide, je renvoie une chaîne de caractères ne comportant que le caractère Unicode 2055 qui représente le symbole ∅. Si la chaîne n'est pas vide, j'utilise un itérateur sur l'ensemble pour le parcourir et ajouter la représentation sous forme de chaîne de caractères de chacun de ses éléments à la représentation de l'ensemble. J'utilise ici un itérateur plutôt qu'une boucle **for**, car je veux profiter de la méthode `hasNext` de `Iterator` qui me permet de savoir si je suis sur le dernier élément de la structure et ainsi de ne pas ajouter de « , » après le dernier élément.
La méthode de comparaison entre ensembles est très basique : l'ordre proposé est fondé sur le cardinal des ensembles à comparer. On aurait pu trouver un ordre plus compliqué sans problème.

Vous remarquerez que l'on aurait pu utiliser les méthodes de `ArrayList` mises à notre disposition : par exemple, `addAll` permet d'ajouter directement une collection dans l'instance de `ArrayList` (cf. méthode `union`). Il faut toujours consulter la javadoc des collections pour savoir si une méthode n'existe pas déjà ! Je ne l'ai pas utilisée ici, parce que `addAll` ne vérifie pas qu'un élément existe déjà dans la liste.
Si l'on réfléchit aux structures de données qui auraient pu typer l'attribut stockant effectivement les éléments, on aurait pu également s'intéresser aux deux collections suivantes :
– `java.util.PriorityQueue` qui représente une queue, une structure servant habituellement à stocker des éléments de façon temporaire avant un traitement. L'intérêt de `PriorityQueue` est que les éléments de la queue sont stockés de façon *ordonnée* en utilisant l'ordre naturel des éléments (la méthode `compareTo` si le type des éléments est un sous-type de `Comparable`). Si la méthode `getMin` est très souvent appelée sur l'ensemble, cette structure peut être intéressante, car elle évite de parcourir toute la liste à chaque appel à `getMin`, il suffit de prendre le premier élément de la `PriorityQueue`, on est garanti que c'est le plus petit.
– `java.util.HashSet` qui représente un ensemble (implanté par une table de *hash*). L'utilisation d'une telle structure de données nous garantit que lorsque l'on ajoute un élément dedans, il n'apparaîtra qu'une seule fois. Cela aurait évité la vérification effectuée dans `add` par exemple et `HashSet` correspond vraiment à ce que l'on veut faire. Par contre, un ensemble n'est pas une liste ou une séquence, donc il n'y a pas de méthode dans `HashSet` pour récupérer le premier élément. On aurait été obligé dans `getMin` d'utiliser un itérateur :

```
public T getMin() throws EnsembleVideException {
    if (this.getCardinal() == 0) {
        throw new EnsembleVideException("L'ensemble est vide !");
    }

    Iterator<T> it = this.ens.iterator();
    T min = it.next();
    T aux = null;

    while (it.hasNext()) {
        aux = it.next();
        if (min.compareTo(aux) > 0) {
            min = aux;
        }
    }

    return min;
}
```

5 Implantation d'une classe `TestEnsemble`

Créer une classe `TestEnsemble` permettant d'exécuter un scénario qui :
– créé un ensemble d'objets de type `Double` place 5 objets de type `Double` dedans ;
– créé un ensemble d'objets de type `Double` place 2 objets de type `Double` dedans ;
– affiche les minima de chacun des ensembles ;
– construit l'union des deux ensembles et affiche son minimum.
On construira de plus une méthode statique `somme` prenant en paramètre un ensemble d'objets de type `Number` et renvoyant la somme des éléments (on utilisera la méthode `doubleValue` de `Number`). On vérifiera que la valeur renvoyée en utilisant le premier ensemble est correcte.

Solution :
Rien de bien particulier ici. On peut juste noter que dans la méthode `somme`, j'utilise comme type paramétré ? **`extends`** `Number` même si `Number` ne réalise pas `Comparable`. Cela ne pose pas de problème, car `Number` est abstraite et on ne pourra jamais déclarer une liste contenant des objets dont le type paramétré est `Number` : ce sera obligatoirement une liste paramétrée avec un type `Integer`, `Double` etc. Un exemple (en commentaire) vous est donné dans le programme de test. J'essaye d'instancier `Ensemble<Number>`. Le compilateur détecte alors l'erreur.
J'ai utilisé un itérateur pour parcourir l'ensemble afin de vous montrer que le type que l'on utilise pour l'itérateur est un *wildcard*. Vous remarquerez que l'on ne peut pas utiliser `Number` dans le type de l'itérateur. On est alors obligé d'utiliser un *wildcard* pour le type de l'itérateur. On pouvait également utiliser simplement une boucle for car `Ensemble` réalise `Iterable` :

```
    for (Number n : ens) {
        res += n.doubleValue();
    }
```