TP de Programmation autour de structures d'ensemble

1 - Ensembles d'entiers

Dans cette première partie, nous allons implémenter une classe permettant de stocker un ensemble non ordonné de valeurs entières. Les ensembles que nous souhaitons manipuler sont en réalité des multi-ensembles (doublons possibles) car nous ne souhaitons pas spécialement assurer l'unicité des éléments. nous les désignerons toutefois par le terme "ensemble" dans toute la suite. Pour démarrer, l'interface EnsembleEntiers vous est donnée. Comme vous pouvez le constater, nos ensembles vont supporter deux opérations, l'ajout d'une nouvelle instance d'un élément et la suppression de la première occurence d'un élément. La suppression d'un élément lèvera une exception lorsque l'élément à supprimer n'est pas trouvé dans l'ensemble. Vous pouvez constater que, pour lever une exception vérifiée, une méthode doit déclarer qu'elle est susceptible de le faire à l'aide du mot clé `throws` suivi du type d'exception. Cela est fait dans l'interface et devra également être fait dans les implémentations.

1.1 - Représentation sous forme de liste chaînée

Nous commencerons par implémenter nos ensembles par une structure de liste chaînée, qui a l'avantage d'être simple à réaliser. L'insertion se fera par exemple en tête, tandis que la suppression devra chercher l'élément et le supprimer, ou lever une exception fabriquée pour l'occasion :

```
throw new Exception(entier + " non trouvé");
```

Ecrivez une classe nommée `EnsembleListeEntiers` contenant cette solution. Vous pouvez utiliser le programme TP1a pour tester votre solution. Si vous voulez un affichage plus verbeux, pensez à définir une méthode publique `toString` renvoyant la représentation textuelle de votre ensemble.

1.2 - Représentation sous forme de tableau

Le défaut d'une structure de liste chainée est que son parcours est relativement inefficace : les éléments contigus ne sont pas adjacents en mémoire et, en raison du chaînage, l'espace qu'elle occupe n'est pas compact. Nous allons donc implémenter une autre version de l'ensemble en utilisant un tableau pour stocker les éléments. Pour rester efficace lors des insertions et des suppressions, nous souhaitons éviter de décaler les éléments en mémoire, nous conserverons donc les éléments non triés. L'insertion se fera par exemple à la fin, tandis que la suppression pourra se faire par échange avec le dernier élément (ou lever une exception le cas échéant). La principale difficulté sera de gérer le cas ou l'espace disponible dans le tableau n'est pas suffisant. Dans ce cas, nous allouerons un nouveau

tableau du double de la taille du tableau existant et y copierons les éléments déjà présents dans l'ensemble. Cette opération est naturellement inefficace, mais le fait de prendre le double de la taille du tableau existant permet d'atteindre une taille suffisante en un nombre de redimensionnements logarithmique en la taille maximale de l'ensemble. Ecrivez une classe nommée `EnsembleTableauEntiers` contenant cette solution. Vous pouvez utiliser le programme TP1b pour tester votre solution.

2 - Ensembles génériques

Nous allons ici découvrir la généricité en java. Le problème de nos ensembles créés précédemment est qu'ils ne manipulent que des entiers. Or, nous aimerions disposer d'ensembles nous permettant de manipuler toutes sortes d'objet et, en particulier, des composants graphiques. Pour cela, nous allons rendre nos ensembles génériques.

En java, toute classe ou interface peut être paramétrée par un type de référence inconnu lors de sa compilation. Cela se fait en ajoutant un nom générique, par exemple `T`, entre < et > après le nom de l'interface ou la classe. Le reste de l'interface ou la classe peut alors utiliser `T` pour déclarer des variables contenant des références à des objets de ce type ou réaliser des affectations entres références de ce type. En revanche, il n'est pas possible de créer un objet de la classe `T`, qui n'est pas une classe mais un type générique, ni d'appeler n'importe quelle méthode à partir d'une référence à un objet de type `T`, car pour garantir la correction de l'ensemble, le compilateur considère que, au pire, les objets référencés par `T` sont de la classe `Object`. Cette classe contient, entre autres, les méthodes `toString`, `equals` ou `clone`. Comme nos ensembles ne manipulent que des références avec, éventuellement, des appels à `toString` lors de l'affichage, nous nous contenterons de cela pour l'instant et verrons ultérieurement comment repousser cette limite.

Dans cette partie une mise à jour de l'interface, Ensemble vous est donnée. Cette nouvelle interface est générique. Vous pouvez également constater que la méthode `supprime` n'est plus susceptible de lever une exception : pour simplifier les choses dans les futurs TPs, nous préférons lever une exception non vérifiée lors de la suppression d'un élément absent de l'ensemble. `NoSuchElementException` définie dans la bibliothèque standard est un bon candidat pour cela.

Modifiez vos ensembles pour en faire des versions génériques implémentant la nouvelle interface et nommées respectivement `EnsembleListe` et `EnsembleTableau`. Récupérez ensuite tous les fichiers de l'apnée 1, le fichier contenant la classe Niveau mise à jour et le fichier contenant la classe ChargeurNiveaux demandée lors de l'apnée 1. Avec le fichier contenant la classe `Niveau` mise à jour, les niveaux stockent maintenant tous leurs composants en double, dans deux ensembles, un stocké sous forme de liste chaînée et l'autre sous forme de tableau, puis affiche les deux ensembles lors de l'appel à `nouvelleBalle`. Le programme principal (celui de l'apnée) vous permet de vérifier cela.