TD 8 : Map et Collection

Objectifs pédagogiques : tables associatives, itérateurs, collections.

8.1 Introduction

Interface `Map<K,V>`

L'interface `Map<K,V>` définit des tables associatives de clé de type K et de valeur de type V. A une clé donnée est associée au plus une valeur. Les opérations de base (complexité faible) sur cette structure sont :

- `V get(K key)` qui rend la valeur associée à la clé `key` (ou null si cette clé n'existe pas)
- `V put(K key, V value)` qui crée ou remplace la valeur value associée à `key`.

Par exemple, on peut voir un `Map<String,Integer> lesNotes` associant aux noms d'étudiants leur note comme

- un tableau indicé par les clés : lesNotes["toto"] = 10, lesNotes["titi"] = 18
- ou, alternativement, comme un ensemble de paires <clé,valeur> : {<toto,10>;<titi,18>}

On peut donc itérer sur un Map avec :

1. juste les clés avec `keySet()`,

```
for (String name : lesNotes.keySet()) {
  System.out.print("nom : " + name);
}
```

2. les paires <clé,valeur> avec `entrySet()`,

```
for (Map.Entry<String,Integer> entry : lesNotes.entrySet()) {
  System.out.print("nom : " + entry.getKey() + " note : " + entry.getValue());
}
```

3. juste les valeurs avec `values()`.

```
for (Integer note : lesNotes.values()) {
  System.out.print("note : " + note);
}
```

Utilisez une syntaxe *foreach* autour de ces appels, ils rendent tous des variantes de Collection.

La méthode `entrySet()` rend un ensemble de `Map.Entry<K,V>` munis de `V getValue()` et `K getKey()`.

Cette interface est implémentée par les classes `HashMap<K,V>` et `TreeMap<K,V>` par exemple.

8.2 Multi-ensembles

Un multi-ensemble sur un ensemble E permet de représenter un «paquet» d'éléments de E, où les éléments ne sont pas nécessairement uniques.

Par exemple, sur l'ensemble $E = \{a, b, c\}$ on peut considérer le multi-ensemble $s = 1'a + 2'b$ (1 occurrence de a et 2 occurrences de b).

$\Rightarrow$ Définir une classe `MultiSet<E>` qui implémente un multi-ensemble en s'appuyant par délégation sur une `Map<E,Integer>` qui associe à chaque élément son nombre d'occurrences dans le multi-ensemble.

⇒ Définir un accesseur `getCount(E element)` pour récupérer le nombre d'occurrences d'un élément passé en argument (0 si l'élément n'est pas dans l'ensemble).

⇒ Définir également une opération d'ajout `void add(E element)`, permettant d'ajouter un élément. Avec l'exemple ci-dessus, l'application de `s.add(a)` donne $s = 2'a + 2'b$. Cette opération incrémente le nombre d'occurrences de l'élément s'il est déjà présent, sinon elle crée une nouvelle entrée avec une seule occurrence.

Le test unitaire suivant doit être vérifié par votre implémentation :

```
public void testAdd() {
  MultiSet<String> stats = new MultiSet<String>();
  stats.add("the");
  stats.add("boing");
  stats.add("the");
  assertTrue(stats.getCount("the")==2);
  assertTrue(stats.getCount("boing")==1);
  assertTrue(stats.getCount("bully")==0);
}
```

⇒ Quelle contrainte(s) doit vérifier le type `E` passé en paramètre pour que les choses se passent «bien»?

⇒ Comment écrire la méthode `toString()` de cette classe pour utiliser la notation $(1'a + 2'b)$ introduite plus haut ?

On souhaite compter le nombre d'occurrences des mots dans un texte.

⇒ Que pensez-vous de l'utilisation d'un `MultiSet` pour répondre à ce besoin ? Comparez à d'autres solutions possibles.

8.3 Belote

Pour cet exercice, nous nous plaçons dans le contexte d'une application de jeu de belote.

8.3.1 Des cartes

On souhaite représenter une carte à jouer d'un jeu de belote caractérisée par un couple formé d'un entier (valeur de la carte de 2 à 14 pour l'as) et d'une enseigne parmi PIQUE, COEUR, CARREAU, TREFLE.

On définit donc tout d'abord le type énuméré suivant :

```
package pobj.belote;
public enum Enseigne {
  PIQUE, COEUR, CARREAU, TREFLE
}
```

⇒ En déduire une classe `CarteAJouer` définie dans le package `pobj.belote`.

⇒ Expliquer comment créer la dame de carreau.

⇒ Ajouter les opérations nécessaires pour stocker la carte dans un `MultiSet<CarteAJouer>` défini en partie 1.

8.3.2 Représentation d'une main

Une main de cartes à jouer est un ensemble de cartes. On propose donc d'implémenter en s'appuyant sur une `List` de `java.util`, par exemple une `ArrayList`.

⇒ Proposer une implémentation la plus courte possible de la classe `pobj.belote.MainCartes`.

⇒ Quels sont les défauts de cette implémentation ?

⇒ Proposez une implémentation au contraire par délégation. On déléguera au moins les opérations `add(CarteAJouer)` et `size()`.

⇒ Comment rendre possible la manipulation d'une main avec une boucle *foreach* ?

```
for (CarteAJouer carte : main) {
  System.out.println(carte);
}
```

⇒ Ajouter une factory `static` supportant la création d'un jeu de 32 cartes (tenir en considération que la carte minimale dans le jeu de la belote est 7).

8.3.3 Calcul de points

La belote se joue avec un jeu de 32 cartes (carte minimale le 7). Les points sans atout sont comptés de la façon suivante : l'as vaut 11 points, le dix 10 points, le roi 4 points, la dame 3 points, le valet 2 points et les autres cartes ne valent rien. A l'enseigne d'atout les points sont les suivants : pour le valet 20 points, le neuf 14 points, l'as 11 points, le dix 10 points, le roi 4 points, la dame 3 points et les autres cartes 0 point.

⇒ Ajouter à la classe `MainCartes` une méthode `int decomptePoints(Enseigne atout)` qui prend en paramètres une enseigne correspondant à l'atout et retourne le décompte des points de la main.

8.4 Itérateurs personnalisés

L'interface standard pour l'itération dans les collections est la suivante :

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
}
```

Dans l'api Java[1], l'implémentation de la méthode `remove()` est optionnelle. Comme le demande le contrat de `Iterator`, une exception non-vérifiée `UnsupportedOperationException` standard est lancée si la méthode est invoquée.

8.4.1 Itération de cartes

⇒ Donner un extrait de code permettant d'itérer les éléments d'une main de cartes (comme la boucle *foreach* plus haut), en supposant une variable `MainCartes main` et en utilisant un itérateur explicite.

8.4.2 Distributeur de carte

On souhaite implémenter un nouvel itérateur personnalisé proposant un parcours aléatoire, afin de l'utiliser pour distribuer les cartes aux joueurs.

On veut pouvoir écrire une boucle d'itération avec itérateur explicite standard de la forme :

```
Iterator<CarteAJouer> it = main.randomIterator();
while (it.hasNext()) {
  System.out.println(it.next());
}
```

[1] http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html

On l'implémente comme une nouvelle classe générique `RandomIterator<T>` dans `pobj.util`, que l'on instancie au sein d'une nouvelle fonction `randomIterator()` déclarée dans `MainCartes` :

```
public Iterator<CarteAJouer> randomIterator();
```

Le constructeur de la classe `RandomIterator<T>` prend une `Collection<T>` en argument (ici, le tableau de cartes). Le comportement de celui-ci est de construire un arrangement aléatoire d'une copie du tableau des cartes (à l'aide `Collections.shuffle()`) puis de construire un `Iterator` sur cette copie et de déléguer dessus `hasNext()` et `next()` pour choisir la prochaine carte tirée.

⇒ Donner le code du distributeur.