TD et TP n° 5 : Collections, Généricité et Enumérations

I) Modéliser avec les collections

Exercice 1 : Comprendre les collections

On rappelle quelques interfaces de collections importantes :

- `List<E>` : liste d'éléments avec un ordre donné, accessibles par leur indice.
- `Set<E>` : ensemble d'éléments sans doublon.
- `Map<K,E>` : ensemble d'associations (clé dans `K`, valeur dans `E`), tel qu'il n'existe qu'une seule association faisant intervenir une même clé.

Ces interfaces peuvent être composées les unes avec les autres pour définir des types plus complexes.
Exemple : un ensemble de séquences d'entiers se note `Set<List<Integer>>`.
Dans chacune des situations suivantes, donnez un type qui convient pour la modéliser :

1. Données des membres de l'équipe de France de football.
2. Idem, avec en plus leurs rôles respectifs dans l'équipe.
3. Marqueurs de buts lors du dernier match (en se rappelant la séquence).
4. Affectation des étudiants à un groupe de TD.
5. Pour chaque groupe de TD, la « liste » des enseignants.
6. Étudiants présents lors de chaque TP de Java ce semestre.

Exercice 2 : Jeu de cartes

Le jeu de cartes classique se compose de 54 cartes : 52 cartes dans 4 familles correspondant à 4 enseignes différentes (pique, cœur, carreau, trèfle), chacune composée de 13 valeurs (as, deux, trois, ... neuf, dix, valet, dame, roi), ainsi que 2 jokers.

1. Pour le jeu à 52 cartes (sans joker), proposez une modélisation objet utilisant les énumérations, permettant de représenter toutes les cartes de ce jeu de cartes. Pour une carte donnée, il faut être capable de récupérer :
   - son enseigne (pique/cœur/carreau/trèfle, appelée souvent "couleur", mais pour éviter les confusions, nous éviterons ce terme)
   - sa valeur
   - sa couleur (rouge ou noir)
   - l'information si cette carte une figure (valet, dame, roi) ou pas

   Essayez de faire en sorte de placer le plus d'implémentation possible au sein des `enum`, quitte à ce que les méthodes des cartes se contentent d'appeler celles des `enum`.
2. Adaptez vos classes (et peut-être même la structure de sous-typage) afin que les classes directement instanciables soient immuables.
3. Proposez une modélisation pour ajouter les jokers (il faut que les cartes "normales" soient instances d'un même type `CarteNormale` dont les jokers ne sont pas... mais il faut aussi que toutes les cartes, jokers compris, soient instances du type `Carte`).
4. Rendez le type `Joker` immuable.
   Maintenant, `CarteNormale` et `Joker` sont immuables. Mais est-ce que `Carte` l'est ? Si ce n'est pas le cas, corrigez ce problème (faites une classe scellée).

5. Programmez des méthodes statiques permettant de générer : un jeu de 52 cartes standard (retourné sous la forme d'une collection), un jeu de 32, un jeu de 54 (avec jokers),
6. Même question pour générer une collection contenant plusieurs exemplaires d'un jeu de carte (par exemple, pour jouer au Rami, il faut deux jeux de 54).

`EnumSet<E>` est une implémentation spécialisée de `Set` pour `E` est un type `enum`. Les opérations `add`, `contains` et `remove` de cette implémentation sont plus performantes que celle des autres implémentations. (`EnumSet` utilise un vecteur de bits et n'appelle pas `hashCode()` contrairement à `HashSet`).
`EnumMap<K, V>` (avec `K` est un type `enum`) suit le même principe pour améliorer les performances des méthodes `put`, `containsKey` et `remove`.

II) Utilisation avancée de collections

Le but de cet exercice est d'implémenter un petit système de base de données en mémoire. Dans le model que nous allons adopter :

- Une base de données (`BaseDeDonnees`) contient plusieurs tableaux.
- Chaque tableau (`Tableau`) est définit par son nom et un ensemble de colonnes.
- Une colonne (`Colonne`) Tous les tableaux ont une colonne `"id"` qui permet d'identifier chaque entré/ligne dans le tableau.

Votre implémentation doit être utilisable avec le code suivant :

```
1  public class Main {
2   public static void main(String[] args) {
3    // creation de la base de donnees
4    BaseDeDonnees db = new BaseDeDonnees("UFR Informatique");
5    // definition de tableau etudiants
6    Tableau etudiants = db.ajouterTableau("etudiants");
7    etudiants.ajouterColonne("nom", TypeDonnee.STRING);
8    etudiants.ajouterColonne("prenom", TypeDonnee.STRING);
9    etudiants.ajouterColonne("groupe", TypeDonnee.INT);
10   // insertion de donnees
11   db.inserer("etudiants", List.of("nom", "prenom", 1));
12   db.inserer("etudiants", List.of("Martin", "Marie", 5));
13   db.inserer("etudiants", List.of("Laurent", "Jean", 1));
14   db.inserer("etudiants", List.of("Simon", "Pierre", 5));
15   // recherche de donnees
16   List<Ligne> resultats = db.chercher("etudiants", "groupe", 5);
17   for (Ligne ligne : resultats) {
18    System.out.println(ligne.get("nom") + " " + ligne.get("prenom"));
19   }
20  }
21 }
```

Exercice 3 : Implémentation simple

1. Définir `TypeDonnee` qui permet d'identifier les différents types de données (`INT`, `STRING`...) qu'on peut stocker dans la base de données.
2. Premièrement on implémentera les entrés/lignes comme des dictionnaires (`Map`) :
   - Implémenter la classe `Ligne` tel que elle encapsule un dictionnaire passé au constructeur.
   - Ajouter une méthode `Object get(String nom)` qui retourne la valeur associé à au nom passé en paramètre.

3. Implémenter la classe `Colonne` définit par un nom et un type de donnée `TypeDonnee`.
4. Implémenter la classe `Tableau` :
   - Ajouter une méthode `void ajouterColonne(String nom, TypeDonnee type)` qui ajoute une nouvelle colonne au tableau.
   - Assurer que tous les tableaux ont par défaut une colonne nommé `"id"` de type `TypeDonnee.INT`.
5. Créer la classe `BaseDeDonnees` avec les méthodes :
   - `Tableau ajouterTableau(String nom)` qui crée et ajoute un tableau avec le nom donné à la base de données, et elle retourne l'instance de tableau créé.
   - `Tableau getTableau(String nom)` qui retourne le tableau avec nom s'il existe dans la base de donnée, sinon elle retourne `null`.
6. La méthode statique `List.of(...)` permet de créer une liste de type `List<Object>` qui contient les éléments passés en argument.
   Écrire un méthode `Map<String, Object> preparer(List<Object> elements)` dans la classe `Tableau` qui associe à chaque nom de colonne (les colonnes ordonnées par ordre d'insertion) une valeur dans le tableau `elements` passé en arguments. La méthode `preparer` doit aussi associer à la clé `"id"` une valeur unique.
7. Les lignes insérées doivent être stockées au niveau de la classe `BaseDeDonnees` (la classe `Tableau` stocke seulement les informations relatives au tableau).
   Ajouter la méthode `void inserer(String nom_tableau, List<Object> elements)` à `BaseDeDonnees` qui permet de créer une ligne à partir de `elements` (utiliser la fonction `preparer` de la classe `Tableau`) et la stocker à la base de données.
8. Définir la méthode `List<Ligne> chercher(String nom_tableau, String nom_col, Object valeur)` qui permet de retrouver les lignes dans le tableau dont la valeur de la colonne `nom_col` est égale à `valeur`.
9. Essayer votre implémentation avec le code donné en dessus.

Exercice 4 : Optimisation de choix de collections

La bibliothèque standard de Java vient avec plusieurs implémentation différentes pour les collections `List` (`ArrayList`, `LinkedList`...) et `Map` (`HashMap`, `LinkedHashMap`, `TreeMap`,...). Dans votre implémentation de l'exercice précédent, vous avez utilisé certaines de ces collections. Les tableaux suivants présentent les complexités [1] des certaines méthodes des collections les plus utilisés.

	add	remove	get	contains
ArrayList	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
LinkedList	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

	get	contains
HashSet	$\mathcal{O}(1)$	$\mathcal{O}(1)$
LinkedHashSet	$\mathcal{O}(1)$	$\mathcal{O}(1)$
TreeSet	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

	get	containsKey
HashMap	$\mathcal{O}(1)$	$\mathcal{O}(1)$
LinkedHashMap	$\mathcal{O}(1)$	$\mathcal{O}(1)$
TreeMap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

1. Rappel : $\mathcal{O}(2^n) > \mathcal{O}(n^3) > \mathcal{O}(n^2) > \mathcal{O}(n \log n) > \mathcal{O}(n) > \mathcal{O}(\sqrt{n}) > \mathcal{O}(\log n) > \mathcal{O}(1)$

Sachant qu'en pratique :

- La création de nouveaux tableaux n'est pas assez fréquente et la plupart des tableaux sont créés juste après la création de base de données.
- Le nombre des lignes d'un tableau est plus grand que le nombres de ces colonnes.
- Les opérations d'insertions et de recherche sont largement utilisés.

Revisiter votre implémentation et améliorer le choix de collections que vous avez fait. Expliquer vos choix.