TD 7 Les collections

Les collections sont des éléments importants de la programmation Java. Très souvent, un tableau ne sera pas l'objet adapté pour capturer un ensemble d'éléments. Dans ce TD nous allons premièrement se rappeler les principes de base de l'utilisation des collections. Puis, nous allons progresser aux choix de design : notamment quand est-ce qu'on utilise les collections et quand est-ce qu'on peut utiliser des tableaux.

Pour ce TD vous aurez besoin de la documentation Java ci-jointe. A chaque question le but sera de chercher si l'objet en question peut utiliser des méthodes déjà pré-implémentées en Java.

Exercice 1

Cet exercice aura comme but de vous familiariser avec la classe `HashSet`. Nous sommes dans une classe TD7.

- Rappel : Qu'est-ce qu'une collection ?
- Rappel : Quels types d'éléments peuvent être mis dans une collection ?
- Rappel : Qu'est-ce qu'un HashSet ?
- Rappel : Disons qu'on veut instancier un objet `ensemble` qui stocke des éléments entiers entre 0 et 100. Quel est le type de ce HashSet ?
- Supposez que dans la classe TD7 vous avez une méthode `public static void main(String[] args).` Faites déclarer et instancier cet objet.
- Dans une classe TD7, on a la méthode suivante :

```
public static Integer getRandomInteger(Integer min, Integer max) {
	Integer diff = max - min;
	Random r = new Random();

	return (r.nextInt(diff) + min);
}
```

Cette méthode génère un nombre entier aléatoire entre 0 et 100.
Utilisez cette méthode dans une méthode main de la même classe TD7 pour générer 20 entiers aléatoirement choisis entre 0 et 100 et pour stocker ces éléments dans l'objet `ensemble`.

SOLUTION :

Une collection est une structure de données qui nous permettent de stocker des objets d'un même type. La Java Collection Framework définit le fonctionnement de divers types de collections.

Les éléments d'une collection doivent être des objets (d'un même type). Une collection ne peut pas donc contenir des variables d'un type primitif.

Un HashSet est une collection de type ensemble : il stocke des éléments sans les répéter. De plus un HashSet stocke les éléments dans une ordre aléatoire, et non pas dans l'ordre qu'on les rajoute.

Une collection ne peut pas stocker des éléments d'un type primitif (par exemple int). Il nous faut le type qui encapsule ce type-là, notamment Integer.

Nous pouvons utiliser la syntaxe :

```
final Set<Integer> ensemble = new HashSet<Integer>();
```

Ce n'est pas incorrect d'utiliser la déclaration avec HashSet<Integer>, mais il est plus élégant de déclarer l'objet en utilisant l'interface Set<E>.

Nous allons mettre les 20 nombres aléatoires dans l'ensemble.

```
for(int i = 0; i < 20; i++) {
	ensemble.add(getRandomInteger(0, 100));
}
```

Exercice 2

Dans ce deuxième exercice le but sera de manipuler les ensembles et leurs éléments. Vous pouvez désormais supposer que nous sommes dans une classe TD7, dans une méthode main dans laquelle on a déjà instancié et créé l'objet ensemble tel qu'il est décrit dans l'exercice précédent.

- Comment peut-on vérifier si le nombre 50 existe dans l'ensemble ?

SOLUTION :

```
System.out.println("Notre ensemble contient le nombre 50 : " + ensemble.contains(50));
```

- Ecrivez du code qui affiche l'ensemble `ensemble` et sa taille. Est-ce que cette taille sera la même à chaque exécution de notre programme ?

SOLUTION :

Nous faisons générer 20 éléments aléatoirement et donc cela dépende de la chance si nous aurons, ou non, de doublons. Les doublons ne sont pas stockés, donc la taille de l'ensemble est variable, entre 1 et 20. On affiche l'ensemble et sa taille :

```
System.out.println(ensemble);
System.out.println("La taille de cet ensemble est " + ensemble.size());
```

- Ajoutez le nombre 42 à votre ensemble. Qu'est-ce qui se passe ? Est-ce que vous allez observer le même effet à chaque nouvelle exécution de votre programme ?

SOLUTION :

Nous ne pouvons pas savoir ce qui se passe si on essaie d'ajouter l'élément 42. Si cet entier existe déjà dans l'ensemble, alors l'ensemble ne changera pas. Si l'élément est bien ajouté (il n'existe pas déjà dans l'ensemble), alors la taille de l'ensemble augmente par 1.

```
int tailleAvantAddition = ensemble.size();
ensemble.add(42);
int tailleApresAddition = ensemble.size();
boolean elementRajoute = !(tailleAvantAddition == tailleApresAddition);
System.out.println(ensemble);

System.out.println("L'element 42 a ete bien ajoute : " + elementRajoute);
```

- Rappel : comment peut-on parcourir un `HashSet` ?

SOLUTION :

Les éléments d'un HashSet ne sont pas indexés. Nous allons avoir donc besoin d'une façon d'itérer sur ses éléments. Pour faire cela nous aurons besoin d'un itérateur.

- Ecrivez du code qui vous permet d'afficher : la valeur la plus proche de 100 dans `ensemble` et sa « position » dans le texte affiché lors de l'exercice précédent (le premier, le deuxième, etc.)

SOLUTION :

Ce code peut être écrit dans la méthode main elle-même, mais j'ai préféré d'écrire une méthode séparée, qui est mise dans la classe TD7. Comme cette méthode sera appelée dans une méthode statique (public static void main) il faut la définir en tant que méthode statique également.

```
// cette methode retourne l'Integer de l'ensemble qui est le plus proche de la valeur maximale des elements d'un ensemble

	public static Integer trouverEntierLePlusProche(Integer max, Set<Integer> ensemble) {
		while(!ensemble.contains(max)) {
			max-=1 ;
		}

		return max;
	}
```

Ce code retourne seulement l'entier le plus proche d'une valeur maximale, mais il va falloir trouver sa position dans l'ensemble.

```
	// cette methode retourne la position d'un entier qui existe deja dans l'ensemble , ou -1 s'il n'existe pas
	public static int trouverPosition(Integer cible, Set<Integer> ensemble) {

		boolean trouve = false;
		int position = 1;
```

```
            Iterator<Integer> monIterateur = ensemble.iterator();
            while(!trouve && monIterateur.hasNext()) {
                  if(monIterateur.next().equals(cible)) {
                        trouve = true;
                        return position;
                  }
                  else {
                        position += 1;
                  }
            }

            if (!trouve) {
                  return -1;
            }

            return position;
      }
```

Dans la méthode main il faut appeler les deux méthodes pour trouver l'entier le plus proche de 100 et d'afficher sa position :

```
      Integer cibleEnsemble = TD7.trouverEntierLePlusProche(100, ensemble);
      System.out.println("Le numero le plus proche de 100 est " + cibleEnsemble);
      System.out.println("Il se trouve sur la position " +
TD7.trouverPosition(cibleEnsemble, ensemble));
```

- Ecrivez du code qui fait copier les éléments d'`ensemble` dans un tableau de la même taille.

SOLUTION :

Malheureusement, il n'y a pas de méthode qui fera cela directement pour les objets de type ensemble. Nous devons donc parcourir l'ensemble avec un itérateur, et ajouter les éléments un par un.

```
Integer[] monTableau = new Integer[ensemble.size()];
int index = 0;
Iterator<Integer> monIterateur = ensemble.iterator();
while (monIterateur.hasNext()) {
      monTableau[index] = monIterateur.next();
      System.out.print(monTableau[index] + ", ");
      index++;
}
```

- Comment pourrait-on faire ordonner les éléments de cet ensemble ? (il suffit d'indiquer une méthodologie, je ne vous demande pas d'écrire du code pour le faire)

SOLUTION :

Il y a beaucoup de stratégies qu'on peut utiliser, y compris : bubble sort, merge sort, heap sort, etc.

Exercice 3

Le but de cet exercice est de comparer les méthodes qui existe en Java pour les HashSets avec les possibilités qu'on a pour un objet de type ArrayList.

- Rappel : qu'est qu'une `ArrayList` ?

SOLUTION :

Une ArrayList est un autre type de collection. Les éléments sont indexés et les doublons sont permis.

- Déclarez un objet `liste` de type `ArrayList`. Comment faut-il changer le code de l'exercice 1 pour mettre les 20 entiers aléatoirement choisis dans la variable `liste` ?

SOLUTION :

La seule partie qui change dans le code est la déclaration e la nouvelle collection.

```
final List<Integer> liste = new ArrayList<Integer>();

// remplir la liste
for (int i = 0; i < 20; i++) {
	liste.add(TD7.getRandomInteger(0, 100));
}
```

- Comment peut-on afficher la liste et quelle sera sa taille ?

SOLUTION :

La taille de la liste sera toujours impérativement 20, car une liste permet des doublons.

```
// L'ensemble et sa taille
System.out.println(liste);
System.out.println("La taille de la liste est " + liste.size());
```

- Répétez l'exercice 2 mais cette fois-ci pour la variable `liste` au lieu de l'objet `ensemble`.

SOLUTION :

Pour la plupart les solutions sont les mêmes, sauf que les éléments de la liste sont déjà indexés et on peut récupérer les éléments sur chaque position. On n'aura pas donc besoin des itérateurs. De plus on la méthode toArray qui transforme une ArrayList dans un tableau.

Voici le code pour chercher l'élément 50 :

```
System.out.println("Notre liste contient le nombre 50 : " + liste.contains(50));
```

Pour chercher le nombre le plus proche de 100 nous allons écrire une nouvelle méthode dans la classe TD7 :

```java
public static Integer trouverEntierLePlusProche(Integer max, List<Integer> liste) {
	while(!liste.contains(max)) {
		max-=1;
	}

	return max;
}
```

En revanche, chercher la position de cette valeur dans la liste est beaucoup plus simple. Il faut seulement retourner le premier index de la méthode en utilisant la méthode indexOf. Dans la méthode main de la classe TD7 on écrit donc :

```java
// L'element le plus proche de 100
Integer cibleListe = trouverEntierLePlusProche(100, liste);
System.out.println("Le numero le plus proche de 100 est " + cibleListe);
System.out.println("Il se trouve sur la position " + liste.indexOf(cibleListe));
```

Lorsqu'on essaie d'ajouter l'élément 42, celui-ci sera ajouté à la liste qu'il soit déjà un élément de celle-ci ou non. La taille de la liste augmente par 1. Pour trouver la position où on a ajouté le nombre 42 nous avons deux possibilités : nous pouvons utiliser la méthode indexOf comme avant, ou nous pouvons utiliser lastIndexOf. La différence entre les deux est liée au fait qu'on peut avoir des doublons dans une liste. La méthode indexOf retourne l'index de la première occurrence d'un élément ; or, si nous venons d'ajouter l'élément 42 c'est plus logique de trouver sa dernière occurrence. Celle-ci peut être trouvée en utilisant la méthode lastIndexOf.

```java
// on veut savoir ce qui se passe lorsqu'on ajoute 42
liste.add(42);
System.out.println(liste);

System.out.println("L'element 42 a ete bien ajoute : true ");
System.out.println("Il est sur la position " + liste.lastIndexOf(42));

// transformer en Array
Integer[] monTableauListe = new Integer[liste.size()];
liste.toArray(monTableauListe);

for(int i = 0; i < monTableauListe.length; i++) {
	System.out.print(monTableauListe[i] + ", ");
}
```

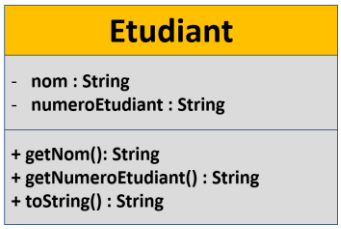
- Quelles sont les différences entre les deux types de collections selon vous ?

SOLUTION :

Je vous laisse tirer vos propres conclusions.

Exercice 4

On suppose que dans un deuxième projet Java on a une classe `Etudiant` tel qu'indiquée par le diagramme de classe ci-dessous :



La méthode `String toString()` retourne le texte : `<numeroEtudiant> : <nom>`. Nous allons supposer maintenant qu'on se trouve dans une autre classe TD7bis, qui a une méthode main.

- Faites instancier un objet `catalogue` de type `ArrayList`, qui stockera les étudiants suivants :
  - Nom : Dupont ; Numéro : 0123
  - Nom : Delafontaine ; Numéro : 1025
  - Nom : Pichon ; Numéro : 5520
  - Nom : Perrin ; Numéro : 0011
  - Nom : Delaune ; Numéro : 1110

- Comment peut-on chercher un étudiant (son nom) à partir de son numéro d'étudiant ?
- Et si on voulait trouver son numéro à partir de son nom ?
- Comment peut-on trouver et afficher tous les numéros d'étudiant alloués ?

SOLUTION :

Nous supposons que la classe Etudiant a déjà été écrite, avec les méthodes indiquées ci-dessus. Nous allons donc mettre en place une liste d'étudiants avec les étudiants indiqués, puis nous allons afficher la liste pour être sûrs qu'elle a bien été instanciée.

```
final List<Etudiant> listeEtudiants = new ArrayList<Etudiant>();
listeEtudiants.add(new Etudiant("Dupont", "0123"));
listeEtudiants.add(new Etudiant("Delafontaine", "1025"));
listeEtudiants.add(new Etudiant("Pichon", "5520"));
listeEtudiants.add(new Etudiant("Perrin", "0011"));
listeEtudiants.add(new Etudiant("Delaune", "9250"));

// afficher la liste
System.out.println(listeEtudiants);
```

Une liste a les éléments indexés par des index entiers : 0, 1, 2, etc. Les ArrayList ont des méthodes qui peuvent trouver un élément à partir de son index et inversement, trouver son index à partir de la valeur de l'élément stocké. Mais, dans ce cas-ci nous voulons trouver un attribut de l'élément à partir d'un autre attribut, et inversement.

Nous écrirons deux méthodes dans ce sens dans la classe TD7 :

```java
// trouver nom etudiant dans un liste par son numero d'etudiant.
public static String trouverEtudiantParNumero(String numero, List<Etudiant> liste) {

	for(int i = 0; i < liste.size(); i++) {
		if(liste.get(i).getNumeroEtudiant().equals(numero)) {
			return liste.get(i).getNom();
		}
	}

	return ("Aucun etudiant avec ce numero !");

}


// trouver numero etudiant dans un liste par son nom.
public static String trouverEtudiantParNom(String nom, List<Etudiant> liste) {

	for(int i = 0; i < liste.size(); i++) {
		if(liste.get(i).getNom().equals(nom)) {
			return liste.get(i).getNumeroEtudiant();
		}
	}

	return ("Aucun etudiant avec ce nom !");

}
```

Nous pouvons faire exécuter soit l'une, soit l'autre de ces méthodes, en fonction de ce qu'on veut trouver. Un exemple est mis ci-dessous. Finalement, pour afficher tous les numéros d'étudiant, on itère sur les index de la liste et on retrouve à chaque fois leurs numéros d'étudiant.

```java
// trouver les noms des étudiants par leur numeros
System.out.println(TD7bis.trouverEtudiantParNumero("0123", listeEtudiants));

// trouver les numéro des étudiants par leur noms
System.out.println(TD7bis.trouverEtudiantParNom("Dupont", listeEtudiants));

// afficher les numeros

for(int i = 0; i < listeEtudiants.size(); i++) {
	System.out.print(listeEtudiants.get(i).getNumeroEtudiant() + ", ");
}
```

Exercice 5

Finalement dans cet exercice nous allons utiliser un objet `mapCatalogue` de type HashMap. Au lieu d'utiliser les objets de type Etudiant qu'on a utilisé lors du dernier exercice, cette fois-ci nous allons utiliser la fonctionnalité d'une `map` pour faire l'association entre le numéro et le nom de l'étudiant.

- Rappel : qu'est-ce qu'une `map` ?
- Rappel : quelles sont les limitations d'une map concernant les clés et les valeurs ?
- Faites déclarer et instancier la mappe `mapCatalogue`
- On veut mettre les 5 noms et numéros d'étudiant de l'exercice précédent dans la collection `mapCatalogue`. Est-ce qu'on peut utiliser les noms en tant que clé pour la mappe ?
- Mettez les 5 noms et numéros d'étudiant de l'exercice 4 dans `mapCatalogue`.
- Refaites le reste de l'exercice 4, mais cette fois-ci en utilisant l'objet `mapCatalogue`.

SOLUTION :

Une map est un autre type de collection, qui fait l'association entre des objets nommés clés et des objets appelés valeurs. Une map peut avoir des valeurs en doublon, mais jamais des clés en doublon.

C'est pourquoi nous ne pouvons pas utiliser les noms en tant que clé : si on fait cela, alors nous ne pourrons jamais avoir deux étudiants avec le même nom. En revanche, les numéros d'étudiant sont uniques, et nous pouvons donc les utiliser en tant que clé.

```
final Map<String,String> mapCatalogue = new HashMap<>();
mapCatalogue.put("0123", "Dupont");
mapCatalogue.put("1025", "Delafontaine");
mapCatalogue.put("5520", "Pichon");
mapCatalogue.put("0011", "Perrin");
mapCatalogue.put("9250", "Delaune");

// afficher la map
System.out.println(maMap);
```

Une des fonctionnalités les plus intéressantes d'une map est la possibilité de retrouver la valeur qui correspond à une clé. Si on veut donc trouver un étudiant par son numéro il suffit d'appeler la bonne méthode :

```
// trouver les etudiants par leurs numéros
System.out.println(maMap.get("0123"));
```

Il est cependant beaucoup plus problématique de trouver une clé par sa valeur. Ce qu'on voudrait faire serait de pouvoir parcourir la collection et regarder l'élément associé à chaque clé pour voir si on peut trouver le bon nom d'étudiant. Le problème est qu'on n'a pas vraiment une façon de parcourir une HashMap.

Par contre, si on regarde la JavaDoc d'une HashMap, nous allons voir qu'il y a une possibilité d'extraire l'ensemble de clés utilisées. Là, c'est mieux puisqu'avec un Set nous savons comment le parcourir. Ce qui donne la méthode suivante.

```
// trouver numero etudiant dans une map par son nom.
public static String trouverEtudiantParNom(String nom, Map<String, String> map) {

	Set<String> lesCles = map.keySet();
	boolean trouve = false;
	String nomTrouve = "";
	Iterator<String> monIterateur = lesCles.iterator();
	while(!trouve && monIterateur.hasNext()) {
		nomTrouve = monIterateur.next();
		if(map.get(nomTrouve).equals(nom)) {
			return nomTrouve;
		}
	}

	return ("Aucun etudiant avec ce nom !");

}
```

Ceci donne la réponse également à la dernière question de l'exercice précédent : pour avoir l'ensemble de clés il suffit d'utiliser la méthode keySet.