amU Faculté des sciences Aix Marseille Université

Année 2025-26

Domaine Sciences et Technologies
Licence Informatique : math-info

Initiation au Génie logiciel : TD 7

Codes UE : SIN5U34

Fusion de listes et gestion de score

1 Fusion de listes

Question 1 : Qu'affiche l'exécution de la méthode `main` code suivant ?

```
public class AlternateListMerger {
    private final List<Integer> list1;
    private final List<Integer> list2;
    public AlternateListMerger(List<Integer> list1, List<Integer> list2) {
        this.list1 = list1;
        this.list2 = list2;
    }
    public List<Integer> getMergedList() {
        List<Integer> result = new ArrayList<>();
        for (int index1 = 0, index2 = 0;
             index1 < list1.size() && index2 < list2.size();
             index1++, index2++) {
            if (index1 < list1.size()) {
                result.add(list1.get(index1));
            }
            if (index2 < list2.size()) {
                result.add(list2.get(index2));
            }
        }
        return result;
    }
    void display() {
        for (int value : getMergedList()) {
            System.out.print(value + " ");
        }
    }
    public static void main(String[] args) {
        List<Integer> list1 = List.of(1, 3, 5);
        List<Integer> list2 = List.of(2, 4, 6);
        ListMerger AlternatelistMerger = new AlternateListMerger(list1, list2);
        listMerger.display();
    }
}
```

Question 2 : On souhaite ajouter une nouvelle façon d'implémenter `getMergedList` qui fusionnerait les deux listes en mettant d'abord tous les éléments de la première liste suivis de tous les éléments de la deuxième liste. Dans le code ci-dessus, quels problèmes de conception identifiez-vous avec ce changement ? À quel(s) principe(s) SOLID les associez-vous ? Justifiez votre réponse.

Question 3 : On souhaite avoir une deuxième classe `ConsecutiveListMerger` dont le seul changement par rapport à `AlternateListMerger` est dans la méthode `getMergedList()` et qui correspondra au code ci-dessous. Proposer une organisation du code sous la forme d'un diagramme de classes UML afin d'éviter la duplication de code entre les classes `ConsecutiveListMerger` et `AlternateListMerger`.

```
public class ConsecutiveListMerger {
    private final List<Integer> list1;
    private final List<Integer> list2;
    public ConsecutiveListMerger(List<Integer> list1, List<Integer> list2) {
        this.list1 = list1;
        this.list2 = list2;
    }
    private List<Integer> getMergedList() {
        List<Integer> result = new ArrayList<>();
        result.addAll(list1);
        result.addAll(list2);
    }
    void display() {
        for (int value : getMergedList()) {
            System.out.print(value + " ");
        }
    }
}
```

Question 4 : Donnez la nouvelle implémentation en Java de la classe `ConsecutiveListMerger` en incluant éventuellement les nouvelles classes et interfaces que vous avez introduites à la question précédente et qui sont utilisées par la nouvelle implémentation de `ConsecutiveListMerger`.

2 Gestion de groupes de coupe du monde

Le but de cet exercice est d'écrire un programme qui calcule le classement des groupes de la coupe du monde de football. Dans cet exercice, nous implémenterons seulement les trois premiers des 8 critères du règlement de la coupe du monde :

- Le plus grand nombre de points sur tous les matchs (règles des coupes du monde jusqu'à 1990 : victoire = 2 points, match nul = 1 point) ;
- La différence de buts sur tous les matchs (différences de buts = buts marqués - buts encaissés) ;
- Le plus grand nombre de buts marqués sur tous les matchs.

En d'autres termes, on regarde en premier critère les nombres de points pour départager deux équipes. Si les deux équipes ont le même nombre de points, on regarde en second critère les différences de buts des deux équipes. Si les deux équipes ont la même différence de buts, on regarde en troisième critère les nombres de buts marqués. Vous trouvez ci-dessous un exemple de classement respectant l'ordre :

Pays	Victoires	Nuls	Défaites	Points	Buts pour	Buts contre	Diff. buts
USA	2	1	0	5	7	3	4
Italie	2	1	0	5	6	2	4
Chili	0	1	2	1	2	4	-2
France	0	1	2	1	1	7	-6

Nous souhaitons pouvoir classer les pays de la façon suivante :

```
public class Main {
    public static void main(String[] args) {
        Team chili = new Team("Chili");
        Team italy = new Team("Italie");
        Team france = new Team("France");
        Team usa = new Team("USA");
        Team[] teams = { chili, italy, france, usa };
        Match[] matches = {
                new Match(usa, italy, 2, 2),
                new Match(usa, chili, 2, 1),
                new Match(usa, france, 3, 0),
                new Match(italy, chili, 1, 0),
                new Match(italy, france, 3, 0),
                new Match(chili, france, 1, 1)
        };
        Group group = new Group(teams, matches);
        group.sort();
        for (Team team : group.teams())
            System.out.println(team.name());
    }
}
```

Les classes `Team`, `Group` et `Match` sont déjà implémentées à l'exception de la méthode `sort` de la classe `Group`.

```
public class Group {
    private final Team[] teams;
    private final Match[] matches;

    public Group(Team[] teams, Match[] matches) {
        this.teams = teams;
        this.matches = matches;
    }

    public void sort() {
        // TODO : implements sort
    }
}
```

```java
public record Team(String name) {
}

public record Match(Team team1, Team team2, int goalCount1, int goalCount2) {
    public boolean isWonBy(Team team) {
        return (team1() == team && goalCount1() > goalCount2())
                || (team2() == team && goalCount1() < goalCount2());
    }
    public boolean isADrawWith(Team team) {
        return (team1() == team || team2() == team) && goalCount2() == goalCount1();
    }
    public int pointsFor(Team team) {
        if (isWonBy(team)) { return 2; }
        if (isADrawWith(team)) { return 1; }
        return 0;
    }
    public int goalDifferenceFor(Team team) {
        if (team.equals(team1())) { return goalCount1() - goalCount2(); }
        if (team.equals(team2())) { return goalCount2() - goalCount1(); }
        return 0;
    }
    public int goalCountFor(Team team) {
        if (team.equals(team1())) { return goalCount1(); }
        if (team.equals(team2())) { return goalCount2(); }
        return 0;
    }
}
```

Question 5 : Quelles parties du code doivent être modifiées pour ajouter une nouvelle méthode de calcul des points qui donne 3 points pour une victoire (et toujours 1 point pour un nul et 0 pour une défaite) ? Quel principe SOLID n'est pas respecté d'après vous ? Justifiez votre réponse.

Afin de permettre de choisir entre les deux méthodes de calcul des points (2 ou 3 points pour une victoire), on introduit l'interface suivante :

```java
public interface ScoreFunction {
    int score(Team team, Match match);
}
```

La méthode de calcul des points ne sera plus dans la classe `Match` (plus de méthode `pointsFor` dans la classe `Match`) mais dans une classe implémentant `ScoreFunction` qui sera utilisé dans `Group`.

Afin de créer des objets calculant les points d'une équipe (qui seront des instances de classes implémentant `ScoreFunction`), on utilisera l'interface suivante qui sera implémentée par deux classes `PointsFunctionOldRuleFactory` (calcul avec 2 points pour les victoires) et `PointsFunctionNewRuleFactory` (calcul avec 3 points pour les victoires) :

```
public interface PointsFunctionFactory {
    ScoreFunction createPointsFunction();
}
```

L'objectif est de permettre l'exécution du code suivant :

```
Match match = new Match(italy, chili, 1, 0);
ScoreFunction pointsFunction1 = new PointsFunctionNewRuleFactory().createPointsFunction();
System.out.println("New score: " + pointsFunction1.score(italy, match));
// affiche New score: 3
ScoreFunction pointsFunction2 = new PointsFunctionOldRuleFactory().createPointsFunction();
System.out.println("Old score: " + pointsFunction2.score(italy, match));
// affiche Old score: 2
```

Question 6 : Écrivez le code des classes `PointsFunctionOldRuleFactory` et `PointsFunctionNewRuleFactory` ainsi que le code de la ou les classe(s) des objets créés par les deux versions de la méthode `createPointsFunction`.

On considère l'interface `Comparator<T>` ci-dessous qui permet de comparer deux éléments de type `T`.

```
public interface Comparator<T> {
    /**
     * Compares its two arguments for order.  Returns a negative integer,
     * zero, or a positive integer as the first argument is less than, equal
     * to, or greater than the second.<p>
     *
     *
     * @param o1 the first object to be compared.
     * @param o2 the second object to be compared.
     * @return a negative integer, zero, or a positive integer as the
     *         first argument is less than, equal to, or greater than the
     *         second.
     */
    int compare(T o1, T o2);
}
```

On souhaite créer une classe `ScoreComparator` qui implémente l'interface `Comparator<Team>` afin de pouvoir comparer les équipes.

Cette classe devra posséder :

- un constructeur qui prend un tableau de matchs (de type `Match[]`) et une instance de `ScoreFunction` en paramètre ;
- implémente la méthode `compare` de sorte à retourner pour deux équipes t_1 et t_2 la valeur de la formule suivante :

$$\sum_{m \in M} (s(t_2, m) - s(t_1, m))$$

avec m le tableau de matchs et s la fonction calculée par la méthode `score` l'instance de `ScoreFunction` donnée en paramètre au constructeur.

L'objectif est de permettre l'exécution du code suivant :

```
public class Main {
  public static void main(String[] args) {
  // Code de création des Team et de Matches
    ScoreFunction pointsFunctionOldRule =
        new PointsFunctionOldRuleFactory().createPointsFunction();
    ScoreComparator scoreComparatorOld = new ScoreComparator(matches, pointsFunctionOldRule);
    System.out.println("Comparison USA France (old) : "+scoreComparatorOld.compare(usa, france));
    // affiche Comparison USA France (old) : -4
    ScoreFunction pointsFunctionNewRule = new PointsFunctionNewRuleFactory().createPointsFunction();
    ScoreComparator scoreComparatorNew = new ScoreComparator(matches, pointsFunctionNewRule);
    System.out.println("Comparison USA France (new) : " + scoreComparatorNew.compare(usa, france));
    // affiche Comparison USA France (new) : -6
  }
}
```

Question 7 : Écrivez le code de la classe `ScoreComparator`.

On souhaite pouvoir composer plusieurs critères de comparaisons. Pour cela, on va introduire une classe `CompositeComparator<T>` qui implémentera l'interface `Comparator<T>`. Cette classe aura une référence d'une liste de `Comparator<T>` et implémentera la méthode `compare(T t1, T t2)` de façon à retourner la valeur `c.compare(t1, t2)` du premier comparateur `c` dans la liste donnant une comparaison différente de `0` (ou `0` si tous les comparateurs donne une comparaison à 0).

Question 8 : Écrivez le code de la classe `CompositeComparator<T>`.

Il reste à écrire l'implémentation de la méthode `void sort()` de la classe `Group` qui trie le tableau `teams` en considérant les matchs du tableau `matches` et en composant les fonctions de score suivantes dans cet ordre :

— nombre de points avec l'ancienne règle des points (2 points par victoire et 1 par nul) ;
— différence de buts (`goalDifferenceFor`) ;
— nombre de buts marqués (`goalCountFor`).

Vous pouvez utiliser la méthode statique `Arrays.sort(T[] array, Comparator<T> comparator)` de Java qui trie le tableau `array` en respectant le comparateur `comparator`.

Question 9 : Écrivez le code de la méthode `sort` de la classe `Group`.

On souhaite pouvoir classer les groupes selon la nouvelle règle de calcul des points (3 points par victoire et 1 par nul).

Question 10 : Décrivez une réorganisation du code permettant de changer la règle de calcul des points en modifiant uniquement une seule ligne de la méthode `main`.