R2-01-03 : TP 4

Développement orienté objets & Qualité de développement

Pour commencer

Continuer dans le projet crée pour les premiers TPs, pour se faire **créer** un package tp4.

Continuer à utiliser les outils mis à votre disposition vu en R1.01 et au début de ce module : les mécanismes automatiques dans l'IDE, le deboggueur, etc.

Exercice 1 : Suppression d'un élément dans une liste

Objectifs R2-03 : utiliser une **`Iterator`** pour effectuer un parcours et une suppression dans une liste.

Cet exercice porte sur la compréhension des listes et des éléments associés à cette liste en mémoire (voir cours associé).

Exercice 1.1 : Suppression d'un élément en utilisant for et while

Dans le paquetage **`tp4`**, créer le package **`collection`** et les classes **`TestIterator`** et **`CollectionUtilitaire`**.

Ajouter à la méthode **`main`** de la classe **`TestIterator`** le code ci-dessous et compléter les instructions demandées :

1. Enlever les entiers inferieurs à 10 dans la première liste d'entiers en utilisant un FOR
2. Enlever les entiers inferieurs à 10 dans la deuxième liste d'entiers en utilisant un WHILE

Le résultat obtenu en utilisant la méthode **`remove`** de la classe **`ArrayList`** dans le cadre de ce « filtre » donne une réponse fausse (un entier en dessous de 10 est conservé dans les listes), voir trace ci-après.

POURQUOI le résultat ne correspond pas à votre attendu ? Lancez un deboggueur pour saisir le problème ! **Aide** : regarder l'indice i de parcours de vos boucles. Si vous ne comprenez pas demander à votre enseignant.

RAPPEL de R1.01 : les entiers (int) ajoutés à une liste sont « transformés » en objets de type **`Integer`** la classe enveloppe correspondante.

Les classes enveloppes (appelées en anglais classes wrapper) sont associées à chacun des types de base. Aux types de base boolean , byte , short , char , int , long , float et double , sont donc associées des classes Boolean , Byte , Short , Character , Integer , Long , Float et Double.

REMARQUE : les deux listes **premiereListeDEntiers** et **deuxiemeListeDEntiers** ci-dessous « pointeront » vers les mêmes objets en début de test mais ces deux listes seront indépendantes dans l'ajout ou la suppression d'éléments. Un entier enlevé de la première liste sera toujours présent dans la seconde liste. Vous le vérifierez avec le test. Si vous ne comprenez pas, lancez un deboggueur pour regarder l'état de la mémoire avant et après le « filtre ».

TestIterator :

```
// Création d'une liste d'entiers
ArrayList<Integer> premiereListeDEntiers = new ArrayList<>();
premiereListeDEntiers.add(5);
premiereListeDEntiers.add(1);
premiereListeDEntiers.add(10);
premiereListeDEntiers.add(3);
premiereListeDEntiers.add(20);
premiereListeDEntiers.add(5);

// Création d'une deuxième liste d'entiers
ArrayList<Integer> deuxiemeListeDEntiers = new ArrayList<>(premiereListeDEntiers);

//  Affichage de la première liste d'entiers
CollectionUtilitaire.afficheCollection("Affichage de la première liste
d'entiers", premiereListeDEntiers);

// Enlever les entiers inferieurs à 10 dans la première liste d'entiers en
utilisant un FOR
// A COMPLETER

// Affichage de la liste première d'entiers après filtre
CollectionUtilitaire.afficheCollection("Affichage de la liste première
d'entiers après filtre", premiereListeDEntiers);

//  Affichage de la deuxième liste d'entiers
CollectionUtilitaire.afficheCollection("Affichage de la deuxième liste
d'entiers", deuxiemeListeDEntiers);

// Enlever les entiers inferieurs à 10 dans la deuxième liste d'entiers en
utilisant un WHILE
// A COMPLETER

// Affichage de la deuxième liste d'entiers après filtre
CollectionUtilitaire.afficheCollection("Affichage de la deuxième liste
d'entiers après filtre", deuxiemeListeDEntiers);
```

Trace obtenue :

```
Affichage de la première liste d'entiers après filtre
1
10
20
```

Exercice 1.2 : Suppression d'un élément en utilisant un iterator

Pour résoudre ce problème de suppression d'élément dans une liste pour ce cas précis, utilisez un objet de type Iterator.

Compléter la classe **`TestIterator`** en utilisant l'iterator fourni par l'**`ArrayList`** :

1. Créer une troisième liste d'entiers en se basant sur la première
2. Enlever les entiers inferieurs à 10 dans la troisième liste d'entiers en parcourant son iterator

Trace obtenue :

```
Affichage de la troisième liste d'entiers après filtre avec un ITERATOR
10
20
```

Exercice 2 : Table de multiplication

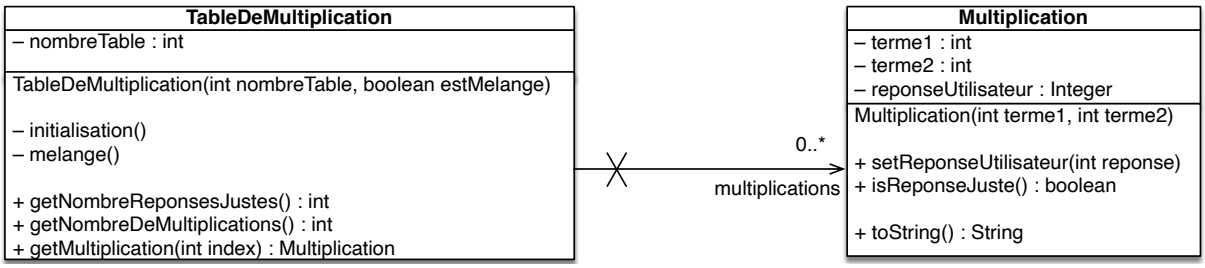
Objectifs R2-01 : rappel Objet, classe, interaction et gestion des erreurs

Vous réaliserez dans la suite une application sur le terminal pour vous exercer aux tables de multiplication. Cette application proposera à l'utilisateur de travailler sur une table entre 1 et 10 et s'exercer soit en mode entrainement (la table est donnée dans l'ordre) soit en mode exercice (la table est donnée dans le désordre), voir trace ci-dessous.

Exercice 2.1 : Mode entrainement et mode exercice

Dans le paquetage **`tp4`**, créer le package **`tabledemultiplication`** et les classes **`TableDeMultiplication`**, **`Multiplication`** et **`TestTableDeMultiplication`**.

Compléter les classes en respectant le diagramme UML ci-après.



Explication de certaines méthodes de la classe `TableDeMultiplication` :

- Le constructeur possède un paramètre **`estMelange`** qui précise si la table va être utilisée pour un mode entrainement ou exercice (ordonnée ou mélangée).
- La méthode **`initialisation`**() initialise la liste de multiplications de la table de multiplication. Pensez à mettre des constantes dans la classe.
- La méthode **`melange`**() mélange la liste de multiplications de la table de multiplication.
- La méthode **`getNombreDeReponsesJustes`**() retourne le nombre de réponses justes données par l'utilisateur.

Explication de certaines méthodes de la classe `Multiplication` :

- L'attribut **`reponseUtilisateur`** est un objet de type **`Integer`** pour avoir la possibilité de le laisser **`null`** au début. Si l'attribut est **`null`**, l'utilisateur n'a pas répondu (un attribut de type **`int`** aurait eu une valeur par défaut de 0). L'utilisation d'un objet dans ce cas précis permet d'avoir une information supplémentaire.
- La méthode **`toString`**() retourne la description de la multiplication « 2 x 5 = »

Compléter la méthode main de la classe **`TestTableDeMultiplication`** pour correspondre à la trace ci-dessous. Pensez à gérer les erreurs de saisies de l'utilisateur, pour le moment seulement des nombres en dehors des attentes.

Trace attendue : les réponses en rouge sont les entrées clavier de l'utilisateur. En gras, la gestion des erreurs de saisies (on redemande à l'utilisateur) :

```
Donner un nombre pour la table de multiplication entre 1 et 10 : 11
Merci de répondre entre 1 et 10 ? 2
Entrainement 1 ou exercice 2 ? 3
Merci de répondre par 1 ou 2 ? 2
Donner les réponses aux multiplications :
2 x 5 = 10
2 x 8 = 16
2 x 4 = 8
2 x 10 = 20
2 x 3 = 6
2 x 7 = 14
2 x 2 = 4
2 x 6 = 12
2 x 1 = 2
2 x 9 = 18
Nombre de réponses justes : 10
```

Exercice 2.1 : Mode sans erreur

Objectifs R2-03 : création et traitement d'une exception

Nous souhaitons rajouter à l'application un mode sans erreur. C'est-à-dire redemander à l'utilisateur une réponse tant que la sienne n'est pas la bonne, voir trace ci-dessous.

Pour ce faire, vous utiliserez le mécanisme d'exception :

- Créer une classe **`ErreurMultiplicationException`**. Cette classe héritera de la classe **`Exception`**. Elle ne contiendra rien mais son nom sera significatif (comme **`NullPointerException`**)
- La méthode **`setReponseUtilisateur`**(…) provoquera une exception de type **`ErreurMultiplicationException`** lorsque la réponse de l'utilisateur ne correspondra pas à la solution.
- L'exception sera « remontée » jusqu'au test pour être traitée.

AIDE :

Modifier les constructeurs de **`TableDeMultiplication`** et **`Multiplication`** pour prendre en compte le mode sans erreur.

Modifier la classe **`TestTableDeMultiplication`** pour ajouter une méthode de demande de réponse utilisateur qui se rappellera en cas d'erreur de calcul de l'utilisateur :

```
private static void demandeReponseUtilisateur(Multiplication  multiplication){

    // Affichage de la multiplication
    System.out.print(multiplication);

    // Demander la réponse utilisateur
    int reponseUtilisateur = entree.nextInt();
    entree.nextLine();

    // Enregistrer la réponse utilisateur
    // Mode sans erreur : attraper l'exception si la réponse de
    // l'utilisateur n'est pas la bonne et redemander la reponse
    // utilisateur
    // A COMPLETER
}
```

Trace attendue : les réponses en rouge sont les entrées clavier de l'utilisateur :

```
Donner un nombre pour la table de multiplication entre 1 et 10 : 2
Entrainement 1 ou exercice 2 ? 2
Mode sans erreur true ou false ? true
Donner les réponses aux multiplications :
2 x 5 = 3
Votre réponse n'est pas correcte, réessayer !
2 x 5 = 10
2 x 1 = 2
2 x 8 = 15
Votre réponse n'est pas correcte, réessayer !
2 x 8 = 16
…
```

Exercice 3 : Les Dalton

Objectifs R2-03 : Enum, interfaces **`Comparable`** et **`Comparator`**.

Dans cet exercice, vous allez décrire les célèbres Dalton et les triés soit en fonction de leur nom, soit en fonction de leur taille, soit les deux !

Dans le paquetage **`tp4`**, créer le package **`dalton`** et les classes **`Dalton`**, **`Taille`**, **`ComparateurNom`**, **`ComparateurTaille`**, et **`TestDalton`**.

Pour créer les classes demandées, voici quelques indications :

- Un Dalton est défini par son nom et sa taille.
- La taille d'un Dalton est de type **`Taille`** un énuméré de valeurs *PETIT*, *MOYEN*, *GRAND*. Voir énumération ci-dessous.
- Un Dalton sait se comparer avec un autre Dalton par sa taille et pour une même taille, par son nom, voir trace attendue ci-après (ordre naturel).

AIDE : vous pouvez comparer les tailles en utilisant simplement l'instruction ci-dessous.

```
taille == Taille.PETIT
```

Compléter les classes demandées pour respecter le test et sa trace associée :

- **`ComparateurTaille`** : classe de comparaison seulement sur la taille

- **`ComparateurNom`** : classe de comparaison seulement sur le nom (solution 1 du cours)
- Dalton.*comparateurNomSolution2 :* Objet de comparaison (classe anonyme) seulement sur le nom (solution 2 du cours)

Enumération d'objets de type `Taille` :

```
public enum Taille {

    PETIT("petit"),
    MOYEN("moyen"),
    GRAND("grand");

    private String libelle;

    private Taille(String libelle) {
        this.libelle = libelle;
    }

    @Override
    public String toString() {
        return libelle;
    }
}
```

`TestDalton`

```
// Création des Daltons
Dalton joe = new Dalton("Joe", Taille.PETIT);
Dalton jack = new Dalton("Jack", Taille.MOYEN);
Dalton william = new Dalton("William", Taille.MOYEN);
Dalton averell = new Dalton("Averell", Taille.GRAND);

// Faire un ensemble triés des daltons sur l'ordre naturel (compareTo)
TreeSet<Dalton> lesDaltons = new TreeSet<Dalton>();
lesDaltons.add(averell);
lesDaltons.add(joe);
lesDaltons.add(william);
lesDaltons.add(jack);

// Affichage
afficheCollection("Les daltons triés par ordre naturel (compareTo) :",
lesDaltons);

// Faire un ensemble triés des daltons avec le comparateur de Taille
(solution 1)
TreeSet<Dalton> lesDaltonsTaille = new TreeSet<Dalton>(new
ComparateurTaille());
lesDaltonsTaille.addAll(lesDaltons);

// Affichage
// ATTENTION 2 daltons avec la même taille donc vu que les doublons ne sont
pas autorisés dans un Set (ensemble)
// Un des daltons ne sera pas conservé
afficheCollection("Les daltons triés par taille :", lesDaltonsTaille);

// Faire un ensemble triés des daltons avec le comparateur de Nom (solution
1)
TreeSet<Dalton> lesDaltonsNom = new TreeSet<Dalton>(new ComparateurNom());
lesDaltonsNom.addAll(lesDaltons);
```

```
// Affichage
afficheCollection("Les daltons triés par nom :", lesDaltonsNom);

// Faire un ensemble triés des daltons avec le comparateur de Nom (solution 2)
TreeSet<Dalton> lesDaltonsNom2 = new
TreeSet<Dalton>(Dalton.comparateurNomSolution2);
lesDaltonsNom2.addAll(lesDaltons);

// Affichage
afficheCollection("Les daltons triés par nom :", lesDaltonsNom2);
```

Trace attendue :

```
Les daltons triés par ordre naturel (compareTo) :
Joe (petit)
Jack (moyen)
William (moyen)
Averell (grand)

Les daltons triés par taille :
Joe (petit)
Jack (moyen)
Averell (grand)

Les daltons triés par nom :
Averell (grand)
Jack (moyen)
Joe (petit)
William (moyen)

Les daltons triés par nom :
Averell (grand)
Jack (moyen)
Joe (petit)
William (moyen)
```