Notation objet, type paramétré, lambda, method reference et calcul paresseux

Exercice 1 - Liste chaînée

On souhaite écrire une liste chaînée, mais contrairement à un TP précédent, ici, nous manipulerons directement les maillons. Chaque maillon est non mutable, il n'est donc pas possible de modifier ni la valeur du maillon ni le chaînage. Donc, la seule façon de construire une liste non mutable consiste à ajouter des maillons *devant* des maillons existants, en construisant la liste à l'envers, le dernier maillon d'abord, puis l'avant-dernier maillon, etc. jusqu'au premier maillon.
Pour éviter d'exposer la valeur `null` à l'utilisateur, on propose que la classe maillon (`Node`) possède les 3 opérations suivantes :

- Un constructeur `Node` qui prend uniquement une valeur en paramètre ;
- Une méthode `prepend` qui prend une valeur en paramètre et crée un maillon dont le successeur est le maillon courant ;
- Une métode `element` qui renvoie la valeur du maillon courant.

On veut de plus que la classe `Node` soit paramétrée par le type des valeurs (`element`) que l'on peut mettre à l'intérieur de la liste chaînée.
Voici un exemple d'utilisation permettant de créer une liste chaînée de maillons ayant les valeurs 1, 2 et 3.

```
var last = new Node<Integer>(3);    // création du dernier maillon
var middle = last.prepend(2);       // on ajoute un maillon devant last avec la valeur 2 et ayant last comme successeur
var first = middle.prepend(1);      // on ajoute un maillon devant middle avec la valeur 1 et ayant middle comme successeur
```

Pour vous aider, nous vous fournissons un série de tests, appelés tests unitaires, qui vérifient la correction de l'implantation des méthodes que vous allez écrire. Ces tests sont reconnus par Eclipse qui les lance dans une fenêtre spécifique.
Pour exécuter les tests, nous avons besoin d'une bibliothèque externe appelée `JUnit 5`, donc dans un premier temps, allez dans les propriétés de votre projet (bouton droit sur le projet, puis Properties en bas du menu) puis sélectionnez Java Build Path, allez dans l'onglet Libraries, sélectionnez Module Path dans l'arbre puis cliquez sur "Add Libraries ...", puis sélectionnez JUnit et enfin JUnit 5. La bibliothèque JUnit est alors ajoutée comme dépendance à votre projet.
Il ne vous reste plus qu'à aller chercher les tests en suivant le lien **vers le fichier `TestNode.java`**. Une fois le fichier téléchargé, vous pouvez faire un copier/coller de tout le code directement dans le répertoire `src` de votre projet dans Eclipse.

1. Écrivez la classe paramétrée et non mutable `Node` dans un package `fr.umlv.node` avec son constructeur et la méthode `element`.
   Vérifiez en lançant les tests unitaires correspondant à la première question (ceux taggés "Q1") que votre implantation est correcte.
2. Ajoutez l'implantation de la méthode `prepend` ainsi qu'une méthode `size` qui renvoie (en temps constant ou O(1) si vous préférez) le nombre d'éléments (le nombre de maillons) de la liste chaînée.
   Vérifiez que les tests taggés "Q2" passent.
3. Écrivez, de façon récursive, la méthode permettant d'afficher les éléments de la liste chaînée séparés par une virgule (et un espace).
   Vérifiez que les tests taggés "Q3" passent.
4. Exécutez le test unitaire nommé `testToStringNoStackOverflow`, quelle est la faiblesse de votre implantation actuelle ?
   Comment corriger le problème sachant qu'il existe en Java une classe `java.util.StringJoiner` (un `StringBuilder` qui sait insérer des caractères de délimitation là où il faut).
   Faites en sorte que votre implantation passe les tests taggés "Q4".
5. On souhaite introduire une méthode `applyFunction` qui prend une fonction en paramètre et permet de créer une nouvelle liste chaînée pour laquelle toutes les valeurs sont calculées en appliquant la fonction à tous les éléments de la liste chaînée actuelle. La liste obtenue contiendra des éléments du même type que la liste initiale.
   Par exemple :

   ```
   var node = new Node<Integer>(32).prepend(128);
   var newNode = node.applyFunction(x -> x * 2);  // créé une nouvelle liste en multipliant les valeurs par 2
   System.out.println(newNode.element());  // affiche 256
   System.out.println(newNode);            // affiche "256, 64"
   ```

   Attention à l'ordre des maillons, `applyFunction` applique la fonction à chaque maillon dans l'ordre de liste pas dans l'ordre de création des maillons.
   En terme d'implantation, le plus simple est de faire un parcours récursif et de créer les maillons en remontant des appels récursifs comme cela on crée les maillons dans le bon sens.
   Il existe dans le package `java.util.function` deux interfaces fonctionnelles qui peuvent représentées la fonction pris en paramètre par `applyFunction`, `Function` et `UnaryOperator`. Expliquer pourquoi ces deux interfaces fonctionnelles peuvent être utilisées pour représenter la fonction prise en paramètre par `applyFunction` puis indiquer selon vous quelle est celle que l'on doit utiliser.
   Faite en sorte que votre implantation passe les tests taggés "Q5".
6. L'implantation actuelle de la méthode `applyFunction` est lente, donc on se propose d'éviter de recréer toute la liste si cela est possible. L'idée est, au lieu de re-créer chaque maillon, de recréer uniquement le premier maillon (on veut toujours être non mutable), de stocker la fonction dans ce maillon, et d'appeler la fonction uniquement lorsque cela est nécessaire.
   En terme d'implantation, cela veut dire qu'un maillon peut avoir ou ne pas avoir de fonction à appliquer. On utilisera `null` (en interne) pour représenter le fait qu'un maillon n'a pas de fonction à exécuter. Il faudra bien sûr faire attention lorsqu'on manipule le champ au fait qu'il puisse ou non être `null`.
   La représentation en mémoire pour `newNode` est

   ```
   | 128        | -> | 32   |
   | x -> x * 2 |    | null |
   ```

   Lorsque l'on parcours la liste chaînée, on applique la fonction à toutes les valeurs des maillons suivants, donc 128 * 2 donne 256 et 32 * 2 donne 64.
   Supposons maintenant que l'on ajoute

```
var newNode2 = newNode.prepend(42).applyFunction(x -> x + 1);
```

La représentation en mémoire pour `newNode2` devient

```
| 42         | -> | 128        | -> | 32   |
| x -> x + 1 |    | x -> x * 2 |    | null |
```

Lorsque l'on parcours la liste chaînée, on applique la fonction à toutes les valeurs des maillons suivants, donc 42 + 1 donne 43, 128 * 2 + 1 donne 257 et 32 * 2 + 1 donne 65.
Comment doit-on modifier la méthode `element` pour qu'elle marche dans le cas où une fonction est stockée dans le maillon ?
Comment doit-on modifier la méthode `prepend` ?
Comment doit-on modifier la méthode `toString` ? Vous pouvez vous aider en introduisant une méthode `compose` qui fait la composition de fonction.
Que doit-on faire si la méthode `applyFunction` est appelée une seconde fois ?
Modifier l'implantation puis vérifier un par un que les tests taggés "Q6" passent.

7. On souhaite ajouter une méthode `forEachIndexed` qui prend en paramètre une fonction à deux arguments (un index et une valeur) et qui appelle la fonction prise en paramètre sur chaque élément de la liste chaînée, avec comme arguments son index dans la liste (en commençant à 0) et la valeur de l'élément correspondant.
   Par exemple :

```
var node = new Node<String>("ramen").prepend("with").prepend("fuel");
node.forEachIndexed((index, element) -> System.out.println(index + " " + element);
```

   affiche "0 fuel", puis "1 with" et "2 ramen".
   Quelle doit être l'interface du package `java.util.function` que l'on doit utiliser pour représenter la fonction prise en paramètre par `forEachIndexed` ?
   Écrivez la méthode `forEachIndexed` et vérifier que les tests taggés "Q7" passent.
8. On cherche à rendre le code de `forEachIndexed` plus efficace car `index` est vu comme un `Integer` au lieu d'être un `int`, donc on paye le coût du boxing (le fait de transformer un int en Integer) à chaque appel de la fonction prise en paramètre de `forEachIndexed`. Pour cela, au lieu d'utiliser une interface fonctionnelle déjà existante dans le package `java.util.function`, créez vous-même votre propre interface fonctionnelle `IndexedConsumer`.
   Modifiez le code de la méthode `forEachIndexed` et vérifiez que les tests taggés "Q8" passent.
9. On souhaite enfin ajouter une méthode `toList` qui renvoie les éléments sous forme d'une liste implantant l'interface `java.util.List`.
   Comment doit on écrire le code de `toList`, sachant que l'on peut ré-utiliser la méthode `forEachIndexed` pour faire le parcours ?
   Écrivez le code de `toList` et vérifiez à chaque fois que les tests marqués "Q9" passent.
10. On peut remarquer que l'on peut aussi ré-écrire `toString` en utilisant `forEachIndexed`.
    Modifiez le code de `toString` et vérifiez que les tests passent toujours.
