Université Lille 1 – Licence Informatique 2011-2012

UE Conception Orientée Objet

TD Listes (Correction)

Étant donné que ce sujet nécessite l'utilisation d'éléments et propriétés très différents des langages orientés objet, et en particulier du langage JAVA, cette correction[1] est découpée en plusieurs parties :

1. la première propose une façon d'implémenter les listes chainées, sans prendre en compte les itérateurs ;
2. la seconde propose une implémentation des itérateurs, sans prendre en compte le ≪*fail-fast*≫ ;
3. enfin, la dernière propose la correction complète du sujet.

1 Première partie de la correction

Cette première partie propose une façon d'implémenter les listes chainées, sans prendre en compte les itérateurs.

1.1 Qu'est ce qu'une liste chainée ?

Une liste chainée est une liste, dont les éléments sont organisés de manière similaire à une chaîne. Chaque élément de la liste (*i.e.* un maillon) est relié à un autre, de sorte que la liste forme une chaine d'éléments.
Le sujet propose une définition récursive dont l'interprétation directe est fournie sur la figure 1. Dans ces exemples, le maillon de tête est le rectangle le plus englobant (par exemple le maillon contenant le chiffre 34 dans la figure 1(c)), et le maillon en queue est le rectangle contenant le texte ≪Liste Vide≫.

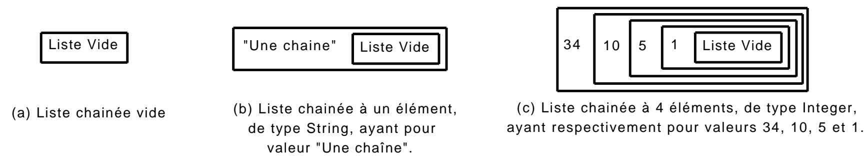


FIG. 1: Illustration de la définition récursive d'une liste chainée. Chaque rectangle correspond à un objet différent, dont la classe est ≪ListeChainée≫.

Une telle représentation est correcte, mais pose toutefois problème pour la définition de listes vides. Cette correction utilise donc plutôt une représentation explicite des maillons (la figure 2 en présente des exemples). Les

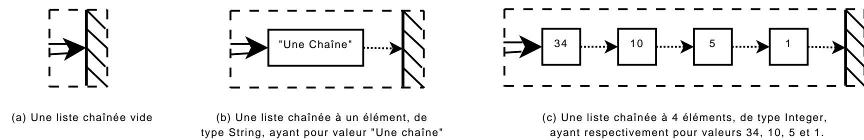


FIG. 2: Illustration de la définition d'une liste chainée faisant apparaître le concept de maillon. Un rectangle en tirets correspond à un objet dont la classe est ≪ListeChainée≫. Un rectangle en traits pleins correspond à un objet dont la classe est ≪Maillon≫.

rectangles pleins y représentent un maillon, le rectangle en tirets y représente la liste. La double flèche le début de liste, la partie droite du rectangle en tirets représente sa fin.

1.2 Identification des différentes classes et attributs

L'observation de la figure 2 permet d'identifier les différentes classes et attributs nécessaires à la création de listes chaînées. En effet, une liste chainée est caractérisée par :

- une tête, qui pointe vers le premier maillon. Pour la représenter, il faut donc ajouter aux instances de la classe ≪ListeChainée≫ un attribut de type ≪Maillon≫ ;
- un ensemble de maillons, qu'il est possible de connaître en suivant la chaîne qui commence au maillon de tête. Ainsi, il n'est pas nécessaire d'ajouter un attribut aux instances de la cclasse ≪ListeChainée≫ pour la représenter ;

[1] Merci à Yoann Kubera pour la rédaction de cette correction

Un maillon est caractérisé par :

- la valeur qu'il contient. Pour la représenter, il faut donc ajouter un attribut aux instances de la classe ≪Maillon≫ ;
- le maillon qui le suit dans la chaîne. Pour le représenter, il faut donc ajouter aux instances de la classe ≪Maillon≫ un attribut de type ≪Maillon≫ ;

Le type de la valeur enregistrée dans les maillons peut *a priori* être n'importe quelle classe. Ainsi, à l'image de l'interface *List* de java, les classes ≪Maillon≫ et ≪ListeChainée≫ seront paramétrées par un **type générique**.

1.3 Classe interne

Un des principes des listes est de pouvoir accéder à leurs éléments sans avoir à savoir comment ces dernières sont implémentées. En effet, à l'aide des méthodes `head()` et `iterator()`, il est possible d'avoir accès à tous leurs éléments. Ainsi, il n'est pas nécessaire que les utilisateurs aient connaissance de l'existence de la classe ≪Maillon≫ (ils n'auront jamais à manipuler eux-mêmes cette classe). Elle sera donc cachée au sein de notre classe ≪ListeChainée≫[2]. Pour cela, elle est déclarée comme une **classe interne** privée.
Le modificateur *privé* implique que la classe ≪Maillon≫ ne pourra être utilisée et manipulée que dans les méthodes déclarées dans la classe ≪ListeChainée≫ (la classe ≪Maillon≫ a une visibilité locale).
Le diagramme UML de notre solution correspond alors à celui de la figure 3.

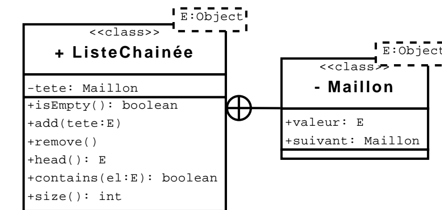


FIG. 3: Diagramme UML de la solution à la première partie du TD.

1.4 Implémentation

Voici le code des différentes classes définies plus haut :

```
1  package mylist;
2  /**
3   * Début de la classe qui permet la représentation
4   * de listes sous forme de listes chaînés
5   */
6  public class ListeChaînée<E>{
7    /**
8     * Déclaration de la classe interne
9     * privée Maillon.
10    */
11   private class Maillon{
12     public E valeur;
13     public Maillon suivant;
14     public Maillon(E val, Maillon suiv){
15       this.valeur = val;
16       this.suivant = suiv;
17       }
18   }
19   // La tete de la liste.
20   private Maillon tete;
21
22   /**
23    * Constructeur créant une liste vide.
24    */
25   public ListeChaînée(){
26     this.tete = null;
27   }
28
29   public boolean isEmpty(){
30     // La liste est vide si elle n'a pas d'éléments, et
31     // donc si ça tête vaut null.
32     return this.tete == null;
33   }
34   public void add(E el){
35     // Création du nouveau maillon de la liste.
36     // Ce maillon sera la nouvelle tête de la liste.
37     // L'élément suivant ce maillon est donc l'ancienne
38     // tete de liste.
39     this.tete = new Maillon(el, this.tete);
40   }
41   public void remove() throws IllegalStateException {
42     if(this.tete == null){
43       // Si la liste est vide, il est impossible d'y
44       // supprimer un élément.
45       throw new IllegalStateException();
46     } else {
47       this.tete = this.tete.suivant;
48     }
49   }
50
51   public E head(){
52     return this.tete.valeur;
53   }
54
55   public boolean contains(E el){
56     Maillon courant = this.tete;
57     while(courant != null){
58       if(courant.valeur.equals(el.valeur)){
59         return true;
60       } else {
61         courant = courant.suivant;
62       }
63     }
64     return false;
65   }
66
67   public int size(){
68     Maillon courant = this.tete;
69     int nombre = 0;
70     while(courant != null){
71       courant = courant.suivant;
72       nombre = nombre + 1;
73     }
74     return nombre;
75   }
76 }
```

[2] Ce principe est aussi utilisé dans la classe java *LinkedList*, pour laquelle la classe du maillon est une classe interne appelée *Entry*

2 Seconde partie de la correction

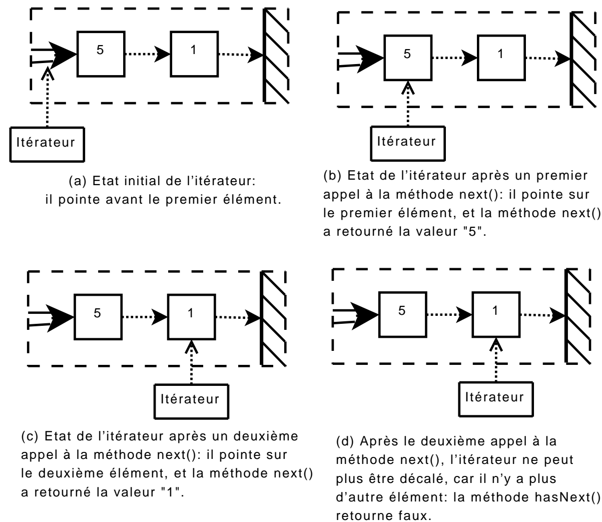
Cette seconde partie propose une implémentation des itérateurs sans prendre en compte le ≪*fail-fast*≫.

2.1 Qu'est ce qu'un itérateur ?

Un itérateur est un objet contenant une ≪flèche≫, qui pointe vers un élément de la liste. Au départ, cette ≪flèche≫ pointe sur la gauche du premier élément de la liste. Les itérateurs permettent de

– *déplacer la ≪flèche≫ vers la droite* (méthode `next()`) ;
– *savoir si la ≪flèche≫ peut encore être déplacée sur la droite* (méthode `hasNext()`).

Une illustration de leur principe de fonctionnement est fournie sur la figure 4.



(a) Etat initial de l'itérateur: il pointe avant le premier élément.

(b) Etat de l'itérateur après un premier appel à la méthode next(): il pointe sur le premier élément, et la méthode next() a retourné la valeur "5".

(c) Etat de l'itérateur après un deuxième appel à la méthode next(): il pointe sur le deuxième élément, et la méthode next() a retourné la valeur "1".

(d) Après le deuxième appel à la méthode next(), l'itérateur ne peut plus être décalé, car il n'y a plus d'autre élément: la méthode hasNext() retourne faux.

FIG. 4: Illustration du mode de fonctionnement d'un itérateur sur une liste chainée.

Bien évidemment, la ≪flèche≫ est implémentée différemment selon les listes :

– une *ArrayList* mémorise des valeurs à l'aide d'un tableau. L'itérateur contiendra donc un nombre entier, correspondant à l'indice dans le tableau ;
– une liste chaînée mémorise les valeurs en les chaînant à l'aide de maillons. L'itérateur contiendra donc un pointeur vers un maillon.

Ainsi, chaque liste dispose de sa propre classe d'itérateur.

2.2 Analyse

L'interface JAVA *java.lang.Iterable* spécifie que la liste dispose d'une méthode itérator. Ainsi, la classe ≪ListeChaînée≫ devra implémenter cette interface. Initialement, la flèche pointe à gauche du premier élément de la liste, ce que nous représenterons par une valeur `null`.
La suppression de l'élément pointé par la flèche (méthode `remove()` de l'itérateur) pose problème. En effet, comment garantir que le maillon qui précédait la flèche ne pointe plus sur la flèche, mais sur le maillon qui la suit ? Pour pallier à ce problème, une deuxième flèche sera utilisée, pointant sur l'élément précédant la flèche dans la liste.
De plus, la méthode `remove()` de l'itérateur ne peut être appelée qu'une seule fois entre deux appels de la méthode `next()`. En effet, d'après la spécification de l'interface *Iterator*, la méthode `remove()` supprime uniquement le dernier maillon pointé par la flèche. Il faudra donc lancer une exception lorsqu'il y a deux appels successifs à cette méthode.

2.3 Classe interne

L'utilisateur n'a, encore une fois, pas besoin de savoir quelle classe concrète implémente l'itérateur, étant donné qu'il n'utilisera que les deux méthodes `next()` et `hasNext()`, fournies par l'interface *Iterator*. L'itérateur dédié à notre classe ≪ListeChaînée≫ sera donc aussi une classe interne[3].

2.4 Implémentation

Voici le code à ajouter au sein de la classe ≪ListeChaînée≫ prenant en compte les itérateurs :

```
package mylist;

import java.lang.Iterable;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * Début de la classe qui permet la représentation
 * de listes sous forme de listes chaînées.
 * L'interface "Iterable" spécifie que la liste dispose
 * d'une méthode iterator().
 */
public class ListeChaînée<E> implements Iterable<E> {

  (...)

  // Methode de la classe "ListeChaînée" permettant de
  // récupérer un itérateur.
  public Iterator<E> iterator(){
          return new ListeChaînéeIterator(this);
  }

  /**
   * Classe d'itérateurs dédiée au type de liste
   * "ListeChaînée".
   */
  private class ListeChaînéeIterator implements Iterator<E>{
     // "Flèche" pointant vers un maillon de la liste.
     private Maillon fleche;
     // "Fleche" vers le maillon qui précède la fleche.
     // Cet élément va être nécessaire dans la méthode
     // remove().
     private Maillon flechePrecedente;

     public ListeChaînéeIterator(){
        // La flèche vaut null lorsquelle pointe à gauche
        // du premier élément de la liste.
        this.fleche = null;
        this.flechePrecedente = null;
     }

     public boolean hasNext(){
        // "ListeChaînée.this" permet de connaître la liste
        // chainée qui a créé l'itérateur.
        if(ListeChaînée.this.isEmpty()){
          // Si la liste est vide, il n'y a pas de prochain
          // élément.
          return false;
        } else {
          if(this.fleche == null){
             // Cas où la flèche pointe à gauche du
             // premier élément, et que la liste n'est
             // pas vide: il y a un prochain élément
             // (le premier de la liste).
             return true;
           } else {
             // Cas où la flèche pointe sur un élément.
             // Il y a un prochain élément si l'élément
             // a un suivant.
             return this.fleche.suivant != null;
           }
        }
     }

     public E next() throws NoSuchElementException {
       // S'il n'y a pas d'élément suivant, l'utilisateur
       // n'a pas le droit d'utiliser la méthode next().
       // On lève donc une exception.
       if(! this.hasNext()){
         throw new NoSuchElementException();
       } else {
         if(this.fleche == null){
           // Cas où la flèche pointe sur la gauche
           // du premier élément. Elle pointe
           // maintenant sur le premier élément.
           this.fleche = ListeChaînée.this.head();
         } else {
           // Sinon, on bouge la flèche sur le
           // prochain élément.
           this.flechePrecedente = this.fleche;
           this.fleche = this.fleche.suivant;
         }
         // On retourne l'élément pointé par la flèche.
         return this.fleche.valeur;
       }
     }

     // Cette méthode supprime l'élément pointé par
     // la flèche.
     public void remove() throws NoSuchElementException {
       // Si la flèche ne pointe vers aucun élément, alors il
       // est impossible de supprimer cet élément: il y a
       // erreur.
       if(this.fleche == null){
         throw new NoSuchElementException();
       } else
          if(this.flechePrecedente.suivant == this.fleche.suivant){
            // Cas où un appel à remove() a déjà eu lieu, et où
            // il n'y a pas eu de nouvel appel à next(): il n'est
            // pas possible d'appeler la méthode remove().
            throw new NoSuchElementException();
       } else {
          // Si l'élément supprimé est le premier de la liste,
          // la tête de la liste change
          if(this.flechePrecedente == null){
            ListeChaînée.this.tete = this.fleche.suivant;
          } else {
            // L'élément précédent pointe vers le suivant
            // de l'élément courant
            this.flechePrecedente.suivant = this.fleche.suivant;
            // La flèche pointe vers l'élément précédent.
            this.fleche = this.flechePrecedente;
          }
       }
     }
   }
}
```

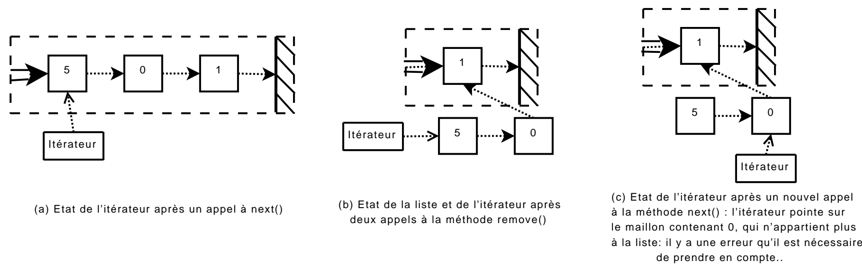
3 Troisième partie de la correction

Cette partie propose une implémentation des itérateurs prenant en compte la fonctionnalité de ≪*fail-fast*≫ des listes.

3.1 A quoi sert le fail-fast ?

La solution proposée ci-dessus fournit toutes les fonctionnalités permettant d'utiliser une liste chaînée. Toutefois, dans certains cas, des erreurs peuvent survenir.
En effet, supposons que l'itérateur pointe sur un maillon de la liste. Que se passe-t-il si ce maillon est retiré de la liste (appel à la méthode `remove()` de la classe ≪ListeChaînée≫) avant le déplacement de l'itérateur ? Dans un tel cas, l'itérateur ne permet plus de se déplacer correctement dans la liste. La figure 5 illustre ce problème.
Le rôle du fail-fast est de détecter ce genre de situations au niveau de l'itérateur et de retourner une exception s'il y a eu modification de la liste entre le moment où l'itérateur est créé, et le moment où sa méthode `next()` est appelée.

[3]Pour information, la classe *LinkedList* de JAVA dispose aussi d'une implémentation interne de l'interface *Iterator*, dont le nom est *ListItr*



(a) Etat de l'itérateur après un appel à next()

(b) Etat de la liste et de l'itérateur après deux appels à la méthode remove()

(c) Etat de l'itérateur après un nouvel appel à la méthode next() : l'itérateur pointe sur le maillon contenant 0, qui n'appartient plus à la liste: il y a une erreur qu'il est nécessaire de prendre en compte..

FIG. 5: Illustration du problème de modification concurrente d'une liste chaînée.

3.2 Implémentation

La méthode la plus simple permettant de savoir s'il y a eu modification de la liste est de vérifier le nombre d'ajouts et suppressions ayant eu lieu entre la création de l'itérateur et l'appel à la méthode `next()`. Pour cela, nous nous basons sur le nombre total de modifications (ajouts et suppressions d'éléments) ayant eu lieu dans la liste depuis sa création.

Ce nombre est mémorisé par chaque itérateur lors de sa création. Si jamais lors d'un appel à la méthode `next()`, le nombre de modifications de la liste est différent du nombre mémorisé par l'itérateur, alors cela signifie que la liste a été modifiée entre temps. L'itérateur n'est donc plus valide : il faudra retourner une exception si il y a utilisation des méthodes `next()` et `remove()`.

De même, un appel à la méthode `remove()` de l'itérateur modifie la liste, qui doit donc mettre à jour son nombre de modifications. L'itérateur ayant fait cette suppression peut adapter ses pointeurs (ses attributs `fleche` et `flechePrecedente`), et donc rester valide. Néanmoins, ce n'est pas le cas des autres itérateurs : ils ne doivent plus pouvoir être utilisés. Pour cela, lors de la suppression d'un élément avec la méthode `remove()` de l'itérateur, il est nécessaire d'incrémenter à la fois le compteur de modifications dans la liste, mais aussi dans celui de l'itérateur ayant effectué la suppression.

Le code complet de la classe ≪ListeChaînée≫ est donc (les parties dont le fond est gris sont des ajouts par rapport aux solutions présentées dans les partie 1 et 2) :

```
1   package mylist;
2
3   import java.lang.Iterable;
4   import java.util.Iterator;
5   import java.util.NoSuchElementException;
6   import java.util.ConcurrentModificationException;
7
8   /**
9    * Début de la classe qui permet la représentation de listes
10   * sous forme de listes chaînés
11   */
12  public class ListeChaînée<E> implements Iterable<E>{
13          /**
14           * Déclaration de la classe interne
15           * privée Maillon.
16           */
17          private class Maillon{
18                  public E valeur;
19                  public Maillon suivant;
20                  public Maillon(E val, Maillon suiv){
21                          this.valeur = val;
22                          this.suivant = suiv;
23                  }
24          }
25          // La tete de la liste.
26          private Maillon tete;
27          // Le nombre total de modifications ayant eu lieu
28          // depuis la création de la liste.
29          private int nombreModifications;
30
31          /**
32           * Constructeur créant une liste vide.
33           */
34          public ListeChaînée(){
35                  this.tete = null;
36                  nombreModifications = 0;
37          }
38
39          public boolean isEmpty(){
40                  // La liste est vide si elle n'a pas d'éléments, et
41                  // donc si ça tête vaut null.
42                  return this.tete == null;
43          }
44
45          public void add(E el){
46                  // Création du nouveau maillon de la liste.
47                  // Ce maillon sera la nouvelle tête de la liste.
48                  // L'élément suivant ce maillon est donc l'ancienne
49                  // tete de liste.
50                  this.tete = new Maillon(el, this.tete);
51                  this.nombreModifications++;
52          }
53
54          public void remove() throws IllegalStateException {
55                  if(this.tete == null){
56                          // Si la liste est vide, il est impossible d'y
57                          // supprimer un élément.
58                          throw new IllegalStateException();
59                  } else {
60                          this.tete = this.tete.suivant;
61                          this.nombreModifications++;
62                  }
63          }
64
65          public E head(){
66                  return this.tete.valeur;
67          }
68
69          public boolean contains(E el){
70                  Maillon courant = this.tete;
71                  while(courant != null){
72                          if(courant.valeur.equals(el.valeur)){
73                                  return true;
74                          } else {
75                                  courant = courant.suivant;
76                          }
77                  }
78                  return false;
79          }
80
81          public int size(){
82                  Maillon courant = this.tete;
83                  int nombre = 0;
84                  while(courant != null){
85                          courant = courant.suivant;
86                          nombre = nombre + 1;
87                  }
88                  return nombre;
89          }
90
91          // Methode de la classe "ListeChaînée" permettant de
92          // récupérer un itérateur.
93          public Iterator<E> iterator(){
94                  return new ListeChaînéeIterator(this);
95          }
96
97          /**
98           * Classe d'itérateurs dédiée au type de liste "ListeChaînée".
99           */
100         private class ListeChaînéeIterator implements Iterator<E>{
101                 // "Flèche" pointant vers un maillon de la liste.
102                 private Maillon fleche;
103                 // "Fleche" vers le maillon qui précède la fleche.
104                 // Cet élément va être nécessaire dans la méthode remove().
105                 private Maillon flechePrecedente;
106                 // Le nombre total de modifications subies par la liste
107                 // lors de la création de l'itérateur.
108                 private int nombreModificationsListe;
109
110                 public ListeChaînéeIterator(){
111                         // La flèche vaut null lorsquelle pointe à gauche
112                         // du premier élément de la liste.
113                         this.fleche = null;
114                         this.flechePrecedente = null;
115                         this.nombreModificationsListe = ListeChaînée.this.nombreModifications;
116                 }
117
118                 public boolean hasNext() throws ConcurrentModificationException {
119                         if(this.nombreModificationsListe != ListeChaînée.this.nombreModifications){
120                                 // Cas où la liste a été modifiée depuis la création de l'itérateur.
121                                 throw new ConcurrentModificationException();
122                         }
123                         // "ListeChaînée.this" permet de connaître la liste chainée qui
124                         // a créé l'itérateur.
125                         if(ListeChaînée.this.isEmpty()){
126                                 // Si la liste est vide, il n'y a pas de prochain élément.
127                                 return false;
128                         } else {
129                                 if(this.fleche == null){
130                                         // Cas où la flèche pointe à gauche du premier
```

```java
131                     // élément, et que la liste n'est pas vide: il y a
132                     // un prochain élément (le premier de la liste).
133                     return true;
134                 } else {
135                     // Cas où la flèche pointe sur un élément.
136                     // Il y a un prochain élément si l'élément a
137                     // un suivant.
138                     return this.fleche.suivant != null;
139                 }
140             }
141         }
142
143         public E next() throws NoSuchElementException, ConcurrentModificationException {
144             if(this.nombreModificationsListe != ListeChaînée.this.nombreModifications){
145                 // Cas où la liste a été modifiée depuis la création de l'itérateur.
146                 throw new ConcurrentModificationException();
147             }
148             // S'il n'y a pas d'élément suivant, l'utilisateur
149             // n'a pas le droit d'utiliser la méthode next().
150             // On lève donc une exception.
151             if(! this.hasNext()){
152                 throw new NoSuchElementException();
153             } else {
154                 if(this.fleche == null){
155                     // Cas où la flèche pointe sur la gauche du
156                     // premier élément. Elle pointe maintenant
157                     // sur le premier élément.
158                     this.fleche = ListeChaînée.this.head();
159                 } else {
160                     // Sinon, on bouge la flèche sur le prochain
161                     // élément.
162                     this.flechePrecedente = this.fleche;
163                     this.fleche = this.fleche.suivant;
164                 }
165                 // On retourne l'élément pointé par la flèche.
166                 return this.fleche.valeur;
167             }
168         }
169
170         // Cette méthode supprime l'élément pointé par
171         // la flèche.
172         public void remove() throws NoSuchElementException {
173             if(this.nombreModificationsListe != ListeChaînée.this.nombreModifications){
174                 // Cas où la liste a été modifiée depuis la création de l'itérateur.
175                 throw new ConcurrentModificationException();
176             }
177             // Si la flèche ne pointe vers aucun élément, alors il est
178             // impossible de supprimer cet élément: il y a erreur.
179             if(this.fleche == null){
180                 throw new NoSuchElementException();
181             } else if(this.flechePrecedente.suivant == this.fleche.suivant){
182                 // Cas où un appel à remove() a déjà eu lieu, et où
183                 // il n'y a pas eu de nouvel appel à next(): il n'est pas
184                 // possible d'appeler la méthode remove().
185                 throw new NoSuchElementException();
186             } else {
187                 // Si l'élément supprimé est le premier de la liste,
188                 // la tête de la liste change
189                 if(this.flechePrecedente == null){
190                     ListeChaînée.this.tete = this.fleche.suivant;
191                 } else {
192                     // L'élément précédent pointe vers le suivant de
193                     // l'élément courant
194                     this.flechePrecedente.suivant = this.fleche.suivant;
195                     // La flèche pointe vers l'élément précédent.
196                     this.fleche = this.flechePrecedente;
197                 }
198                 // Mise à jour du nombre de modifications subies par la
199                 // liste. Puisque cette modification est prise en compte
200                 // par cet itérateur, il faut aussi mettre à jour son attribut
201                 // nombreModifications.
202                 ListeChaînée.this.nombreModifications++;
203                 this.nombreModifications++;
204             }
205         }
206     }
207 } // fin de la classe "ListeChaînée"
```