TD 4 : Objets itérables

Objectifs pédagogiques :

- **collections**
- **itérateurs**

4.1 Itérateur sur une matrice

Nous fournissons la classe `Matrice<T>` qui permet de structurer un nombre fini d'éléments de type `T` sur N lignes et M colonnes. L'implémentation que nous donnons repose sur un tableau Java à deux dimensions.

```
public class Matrice<T> {

  private final T[][] mat;

  // évite un avertissement d'Eclipse sur le type des tableaux
  // (explications détaillées au cours 7 sur les génériques Java)
  @SuppressWarnings("unchecked")
  public Matrice(int nblignes, int nbcolonnes) {
    mat = (T[][]) new Object[nblignes][nbcolonnes];
  }

  public int nbLignes() {
    return mat.length;
  }

  public int nbColonnes() {
    return mat[0].length;
  }

  public T get(int lin, int col) {
    return mat[lin][col];
  }

  public void put(T e, int lin, int col) {
    mat[lin][col] = e;
  }
}
```

Nous rappellons les déclarations des interfaces `Iterable<T>` et `Iterator<T>` du paquetage `java.lang`.

```
public interface Iterable<T> {
   Iterator<T> iterator();
}
```

```
public interface Iterator<E> {
   boolean hasNext();
   E next();
}
```

Question 1. Comment faut-il modifier la classe `Matrice` pour que le code suivant compile ?

```java
public int sum (Matrice<Integer> mat) {
   int cpt = 0;
   for (Integer i : mat) {
      cpt += i;
   }
   return cpt;
}
```

Question 2. Donnez le code de l'itérateur (classe interne à `Matrice`). L'itérateur commence par parcourir la première ligne de la matrice, de gauche à droite, puis la deuxième ligne, de gauche à droite, etc. jusqu'à la dernière ligne, de gauche à droite. Les indices correspondants sont donc, dans l'ordre : $(0,0)$, $(0,1)$, …, $(0,\texttt{nbcolonnes}-1)$, $(1,0)$, $(1,1)$, …, $(1,\texttt{nbcolonnes}-1)$, …, $(\texttt{nblignes}-1,0)$, …, $(\texttt{nblignes}-1,\texttt{nbcolonnes}-1)$.

4.2 Classe `ListVector<T>`

Nous donnons en annexe les deux classes présentées en cours : `MyArrayList<T>` et `MyLinkedList<T>`.

Question 3. Rappelez la différence entre ces deux implémentations :

- en termes de mémoire ;
- en termes de complexité d'accès et d'ajout.

Dans cette question, nous définissons une nouvelle collection, nommée `ListVector<T>`, basée sur une liste chaînée (`MyLinkedList`) de vecteurs (`MyArrayList`), afin de trouver un compromis entre les coûts d'ajout en fin de structure (`add`) et d'accès aléatoire (`get`) des deux structures. Pour cela, une taille maximale de vecteur est fixée une fois pour toutes lors de la construction d'une `ListVector`. Lors d'un ajout dans une `ListVector` :

- si le dernier vecteur de la liste chaînée n'a pas atteint sa taille maximale, l'élément est ajouté à ce vecteur ;
- s'il a atteint sa taille maximale, nous ajoutons à la liste chaînée un nouveau chaînon contenant un nouveau vecteur vide, puis nous ajoutons l'élément à ce vecteur.

Par ailleurs, l'accès à un élément à une position donnée consiste à d'abord chercher le maillon de la liste chaînée contenant le bon vecteur (coût linéaire en le nombre de chaînons), puis à accéder à l'élément dans le vecteur (coût constant).

L'implantation de `ListVector<T>` utilise une `MyLinkedList` de `MyArrayList` d'éléments `T`. Elle s'appuie donc par délégation sur une `MyLinkedList<MyArrayList<T>>`.

Question 4. Implémentez une classe `ListVector<T>` ayant un constructeur prenant en argument un entier représentant la taille maximale des vecteurs, et ayant les méthodes suivantes :

- `int size()` : retourne le nombre d'objets contenus. Nous recommandons de maintenir la taille actuelle dans un attribut entier, comme c'était le cas dans `MyArrayList` et `MyLinkedList`, pour faciliter l'implantation.
- `boolean add(T element)` : ajoute `element` en dernière position. Si, avant l'ajout, le dernier `MyArrayList` est plein, en allouer un nouveau. La méthode retourne toujours `true` (la collection est bien modifiée par l'ajout ; rappelons que ce n'est pas toujours vrai pour les collections qui n'autorisent qu'une seule occurrence de chaque objet, comme les `Set`, et qui retournent `false` sans modifier la collection si l'objet y est déjà présent).
- `T get(int index)` : retourne l'objet en position `index` (compris entre 0 et `size()`-1 inclus). Vous pouvez calculer directement avec des modulos et des divisions les bonnes positions dans la liste chaînée et le vecteur.

Nous voulons maintenant pouvoir itérer sur notre `ListVector` avec une boucle *foreach*.

L'itérateur de cette classe, nommé `ListVectorIterator`, s'appuiera par délégation sur deux itérateurs : un qui itère sur la liste chaînée du `ListVector` et renvoie donc un nouveau vecteur à chaque

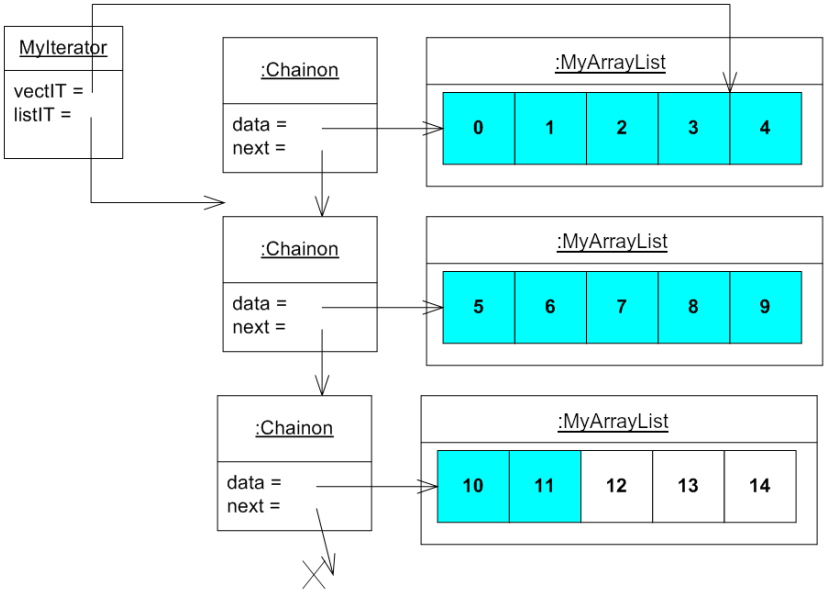

Figure 1: Illustration de l'organisation d'une `ListVector` et de ses itérateurs internes. Les vecteurs sont alloués ici avec une taille de 5. La `ListVector` contient ici 12 éléments (indicés 0 à 11), et le dernier vecteur n'est pas encore plein. L'ajout d'un 16ème élément provoquera l'allocation d'un nouveau chaînon et de son vecteur. Le `ListVectorIterator` (voir question 6) vient de lire ici le quatrième élément (indicé 3) : son `listIt` pointe juste après le premier élément de la liste chaînée et son `vectIt` pointe juste après le quatrième élément du premier vecteur. En Java, les `Iterator` sont toujours *entre* deux éléments.

appel à `next()`, et un qui itère au sein d'un vecteur courant, et pour lequel un appel à `next()` renvoie un nouvel élément. Ceci est illustré dans la figure 1. Nous les appellerons dans la suite respectivement `listIT` et `vectIT`.

Question 5. Donnez le type des deux itérateurs `listIT` et `vectIT`.

Question 6. Réalisez l'itérateur `ListVectorIterator` sur `ListVector`.

À la construction, nous positionnons le curseur `listIT` sur le début de la liste chaînée, et le curseur `vectIT` à vide (`vectIT.hasNext()` retourne immédiatement `false`). Pour positionner à vide, vous pouvez utiliser l'itérateur vide fourni par : `static Iterator<T> Collections.emptyIterator()`.

L'opération `hasNext()` teste si au moins un des deux curseurs n'est pas arrivé au bout de son itération.

L'opération `next()` teste si `vectIT` est au bout. Si c'est le cas, il décale le curseur `listIT` et repositionne `vectIT` au début du prochain vecteur de la liste chaînée. Dans tous les cas, la méthode rend ensuite le `next()` de l'itérateur `vectIT`. Nous supposons donc que le dernier vecteur n'est jamais vide (ce qui est bien assuré par notre implantation de `add`).

Annexe

```java
package pobj.collections;

import java.util.Iterator;
import java.util.NoSuchElementException;

public class MyArrayList<T> implements Iterable<T> {
   private T[] tab;
   private int size = 0;

   @SuppressWarnings("unchecked")
   public MyArrayList(int max) {
      // new T[] est illegal car T est un argument de générique, d'où le new Object[]
      // (ceci sera détaillé au cours 7 sur les génériques)
      tab = (T[]) new Object[max];
   }

   public int size() {
      return size;
   }

   @SuppressWarnings("unchecked")
   public boolean add(T element) {
      if (size == tab.length) {
         T[] tmp = (T[]) new Object[tab.length * 2];
         for (int i = 0; i < tab.length; i++) {
            tmp[i] = tab[i];
         }
         tab = tmp;
      }
      tab[size++] = element;
      return true;
   }

   public T get(int index) {
      if (index < 0 || index >= size)
         throw new ArrayIndexOutOfBoundsException();
      return tab[index];
   }

   public Iterator<T> iterator() {
      return new MyArrayListIterator();
   }

   private class MyArrayListIterator implements Iterator<T> {

      private int index = 0;

      @Override
      public boolean hasNext() {
         return index < size;
      }

      @Override
      public T next() {
         if (index >= size)
            throw new NoSuchElementException();
         return tab[index++];
      }
   }
```

```
}
```

```
package pobj.collections;

import java.util.Iterator;
import java.util.NoSuchElementException;

public class MyLinkedList<T> implements Iterable<T> {

   private Chainon<T> tete = null;
   private Chainon<T> queue = null;
   private int size = 0;

   public boolean add(T element) {
      if (tete == null) {
         tete = new Chainon<T>(element, tete);
         queue = tete;
      } else {
         queue.setNext(new Chainon<T>(element, null));
         queue = queue.getNext();
      }
      size++;
      return true;
   }

   public int size() {
      return size;
   }

   public T get(int index) {
      Chainon<T> cur = tete;
      if (index < 0 || index >= size())
         throw new ArrayIndexOutOfBoundsException();
      for (int i = 0; i < index; i++) {
         cur = cur.getNext();
      }
      return cur.getData();
   }

      public T getLast() {
            return queue.getData();
      }

   @Override
   public Iterator<T> iterator() {
      return new MyLinkedListIterator();
   }

   private static class Chainon<T> {
      private T data;
      private Chainon<T> next;

      public Chainon(T data, Chainon<T> next) {
         this.data = data;
         this.next = next;
      }

      public T getData() {
         return data;
      }
```

```
      public Chainon<T> getNext() {
         return next;
      }

      public void setNext(Chainon<T> next) {
         this.next = next;
      }
   }

   private class MyLinkedListIterator implements Iterator<T> {

      private Chainon<T> cur = tete;

      @Override
      public boolean hasNext() {
         return cur != null;
      }

      @Override
      public T next() {
         if (cur == null)
            throw new NoSuchElementException();
         T data = cur.getData();
         cur = cur.getNext();
         return data;
      }
   }
}
```