Fiche résumé nº 2 – Les itérateurs

1. Introduction

Rappel : dans le TAD Liste précédent on a une série d'opérations disponibles

- estVide()
- entête()
- enqueue()
- succ()
- préd()
- vider()
- …

Après analyse, on peut distinguer deux finalités bien différentes dans ces opérations :

1. Gestion de la SDD (stockage)
2. Système d'énumération (notion d'élément courant dans le parcours)

Bien que ce mélange ait été généralisé très longtemps, désormais le concept d'itérateur s'est imposé pour bien séparer cet ensemble d'opérations en deux nouveaux TAD :

- **Liste** : on réduit ce TAD aux opérations de son unique vocation (gestion du stockage)
- **Itérateur** : on introduit un TAD itérateur (de liste) pour y mettre les autres opérations relatives à l'élément courant
- Et on ajoute une opération pour obtenir un itérateur à partir d'une liste donnée.

L'itérateur est alors un auxiliaire, dont le rôle est de permettre d'énumérer le contenu de la liste.
Analogie avec les tableaux :

```
int i;
for (i=0; i<tab.length; i++) {
    Traitement de tab[i];
}
```

```
Itérateur i = l.itérateur();
for (i.entête(); !i.estSorti(); i.succ()) {
      Traitement de i.valec();
}
```

L'entier est en quelque sorte l'itérateur nécessaire pour énumérer le contenu d'un tableau.
Il conviendra donc de répartir ces opérations dans 2 interfaces Java au lieu d'une : Liste et Itérateur.
N.B. Outre une meilleure modélisation séparant et clarifiant les 2 aspects, l'introduction de la partie itérateur permet de plus d'effectuer plusieurs itérations simultanées sur la même liste.

2. Mise en œuvre (Java)

Tout en offrant à l'extérieur des opérations unifiées, l'itérateur est un objet intimement lié avec la structure interne de la liste. Il faut donc le décliner en rapport avec une mise en œuvre de liste visée.
Nous nous plaçons ici dans le cas d'implémentation de liste simplement chaînée, avec éléments fictifs aux extrémités comme vu précédemment.

```
public class ListeSimplementChaînée implements Liste {
    private Maillon tête;

    private static class Maillon {
          Object val;
          Maillon s;
    }

    // Méthodes implémentant l'interface Liste + constructeur(s) :
```

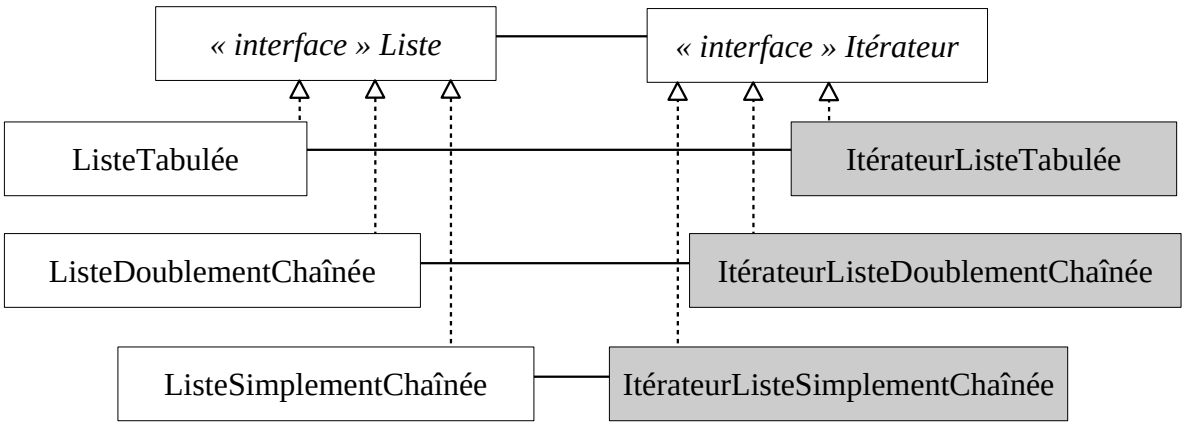
```
    ...
    // Et la méthode pour obtenir un itérateur de cette liste :
    public Itérateur itérateur() {
        return new ItérateurListeSimplementChaînée(this);
    }
}
```

L'implémentation de l'itérateur associé à cette liste contient tout l'outillage lié à l'énumération des éléments :

```
public class ItérateurListeSimplementChaînée implements Itérateur {
    private ListeSimplementChaînée l;
    private Maillon prédec, ec;

    public ItérateurListeSimplementChaînée(ListeSimplementChaînée liste) {
        l = liste;
        entête();
    }

    public void entête() {
        ec = l.tête.s;
        prédec = l.tête;
    }

    public boolean estSorti() {
        return ec==l.tête || ec.s==null; // cas avec 2 maillons d'extrémités
    }

    ...
}
```

3. Remarques et améliorations

a) Visibilité des classes



L'utilisateur extérieur n'a pas besoin d'être parasité par certaines classes purement auxiliaires, notamment la classe effective particulière implémentant l'itérateur de chacune des variantes de listes :

- Il ne connaîtra que l'interface publique Itérateur (abstraction)
- Chaque classe concrète « cachée » (grisée sur le schéma) pourra être une classe interne privée (encapsulation, comme l'a été la classe Maillon).

Ces classes auxiliaires, devenues internes et privées, seront totalement masquées à l'utilisateur externe : non accessibles, invisibles, et non documentées dans la Javadoc.

L'avantage d'utiliser ce concept de classe interne permet de tirer profit adroitement du qualificatif static :

- idéalement la classe interne Maillon sera **`static`** :
  - c'est en fait une classe **auxiliaire de classe** => liée à la classe englobante.

- Idéalement la classe interne ItérateurListeSimplementChaînée ne sera pas `static` :
  - c'est en fait une **auxiliaire d'instance** => liée à un objet de la classe englobante
  - et ainsi l'itérateur aura accès aux attributs de l'instance qui le contient, ce qui évite de mémoriser dans ses propres attributs la liste à laquelle il est associé.

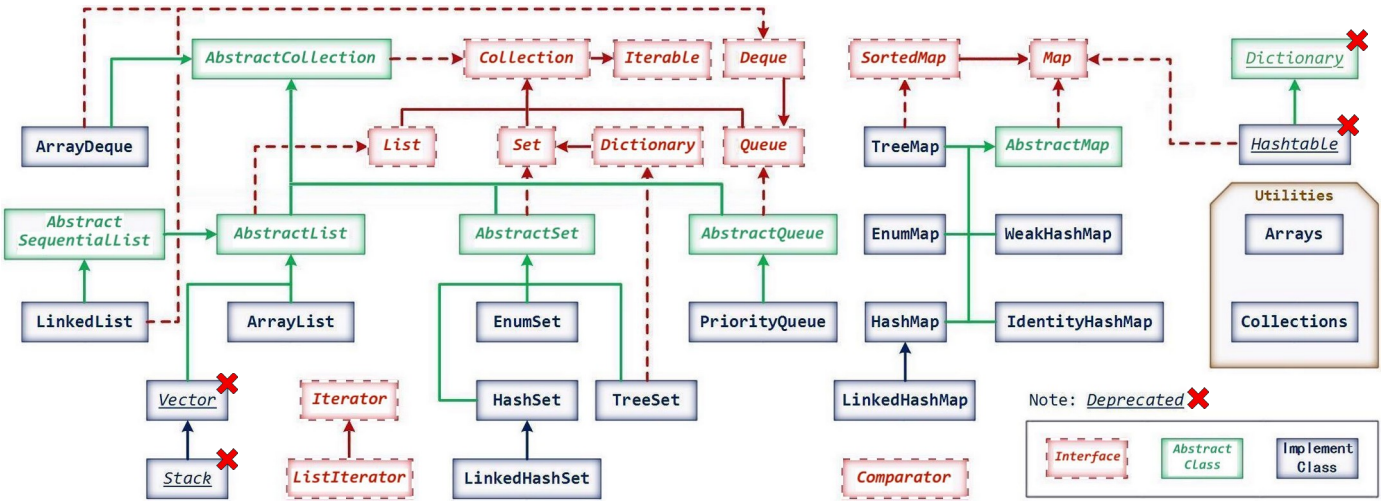
b) Généricité

Les listes devant contenir des éléments du même type, les 2 TAD bénéficieraient d'être exprimés de manière générique :

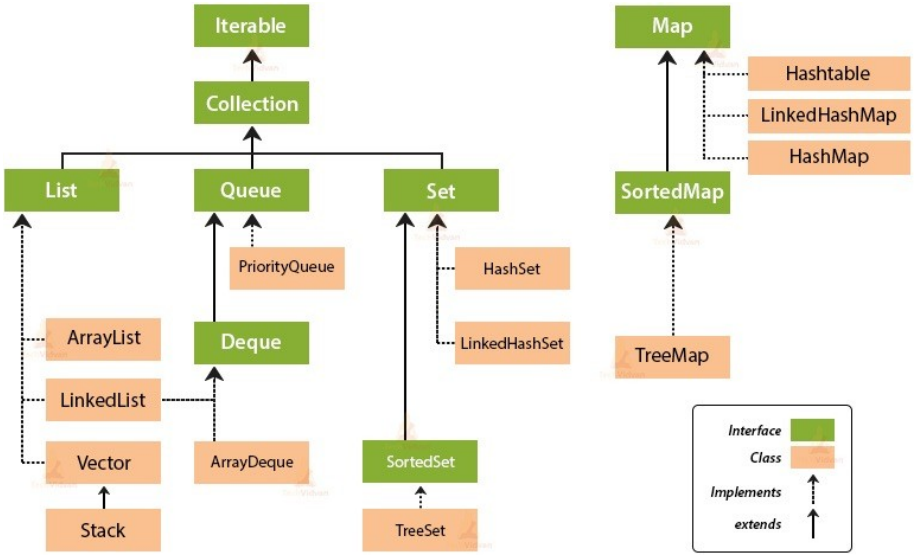
- TAD génériques Liste[T] et Itérateur[T]
- à traduire en utilisant les *generics* de Java : `Liste<T>` et `Itérateur<T>`

4. Le *framework* des collections Java

Les développeurs ayant construit Java lui ont adjoint de nombreuses bibliothèques standards, contenant en particulier le *framework* des collections Java (voir doc officielle[1]) : une mise en place de nombreuses structures de données dont la conception et modélisation ont donné lieu a beaucoup de classes et interfaces :



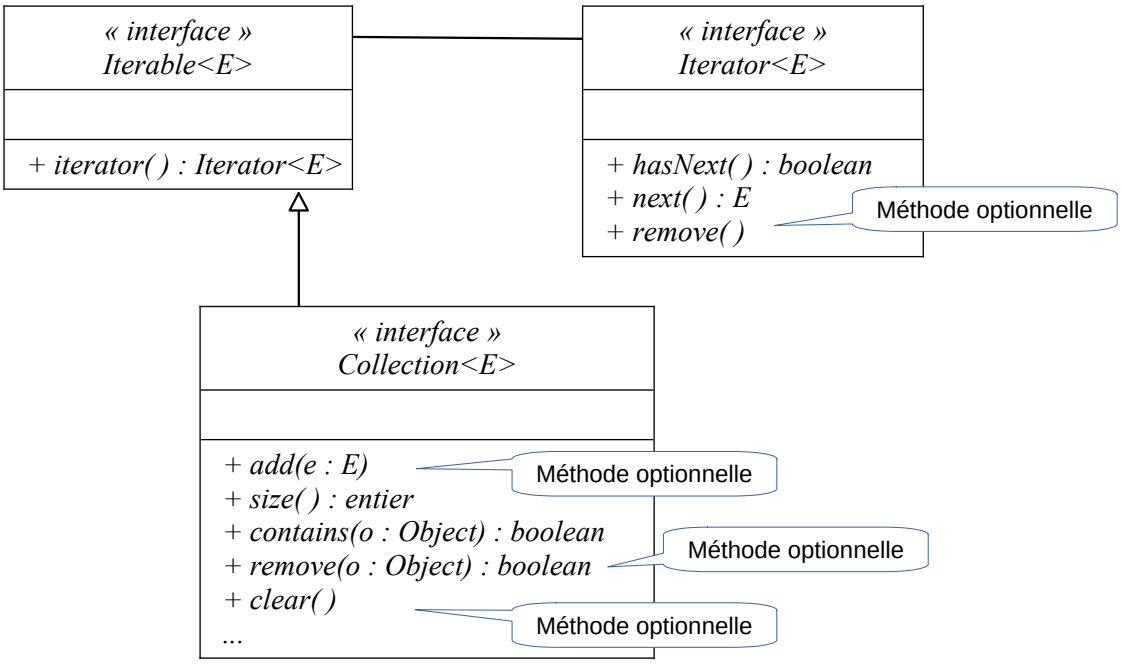
En vue un peu plus résumée mettant en avant l'aspect hiérarchique :



1 Référence : https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html

3 interfaces ont un rôle prépondérant :

- **Collection** : qui impose des méthodes unifiées pour l'usage de base qui seront communes à toutes les collections sous-jacentes
- **Iterator** : l'interface respectée par tout itérateur concret d'une certaine collection concrète
- **Iterable** : qui impose la présence de la méthode pour obtenir un itérateur dans toute collection (et aussi pour les tableaux Java).



Cette propriété Iterable autorise le raccourci d'écriture « *for each* » de Java pour écrire une boucle `for` énumérant les éléments contenus, sans avoir besoin d'expliciter l'itérateur nécessaire qui sera exploité implicitement par Java.

```
for (UnType e : uneCollection) { // « for each e in uneCollection »
      // Utiliser l'élément e
}
```

N.B. Ce *framework* Java met en œuvre les mêmes principes que ceux que nous avons analysés en détails :

- TAD Liste => interface List
- TAD Itérateur => interface Iterator
- mise en œuvre tabulée => classe ArrayList
- mise en œuvre doublement chaînée => classe LinkedList

De plus, quoique apparemment différents, nos mécanismes d'itérateurs sont bien équivalents : Comparons :

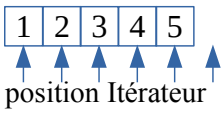
```
void afficherListe(Liste l) {
    Itérateur i = l.itérateur();
    i.entête();
    while (!i.estSorti()) {
          Object o = i.valec();
          System.out.println(o);
          i.succ();
    }
}
```

```
void printList(List l) {
    Iterator i = l.iterator();
    // en tête dans le constructeur
    while (i.hasNext()) {
          Object o = i.next();
          System.out.println(o);
          // avance dans next()
    }
}
```

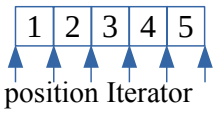
La différence est alors dans la subtilité de l'explication sémantique :

- par clarté technique nous avons considéré que notre élément courant était un curseur indiquant une position d'élément
- l'explication dans la Javadoc de l'API Java présente intellectuellement les choses comme un curseur se plaçant entre 2 éléments (ce qui techniquement n'est toutefois pas le cas, mais très didactique)

Illustration :



position Itérateur



position Iterator