U.B.S. - I.U.T. de Vannes
*Département Informatique*

# R3.02 – TP3 – Classe de type Collection

***J-F. Kamp***

Septembre 2024

Troisième TP d'exercices sur la manipulation de structures assez classiques (listes, arbres) enseignés en seconde année de BUT informatique. Les exercices sont à réaliser en Java, ils abordent également les notions de contrat et de généricité.

Mise en pratique des notions vues en cours

# *TP3 : Construction d'une classe conforme aux normes de l'API Java, la classe Sac de type Collection*

## *1. La classe Sac<E>*

Pour ce TP, vous devez créer un nouveau projet (*projSac* par exemple) sous *Eclipse*. Pour le test (classe *TestSac* qui appartiendra au paquetage *structdonnees*), faire usage obligatoirement de *JUnit4*.

Un "sac" est une collection dans laquelle les données à stocker sont insérées dans un ordre quelconque. Il n'y a pas de notion de clé primaire (au sens base de données), dès lors deux données identiques peuvent co-exister dans la même collection.

La classe *Sac* ressemble à la classe *ListChainee* que nous avons déjà développé (Figure 1) :

- la liste est simplement chaînée (déplacement dans 1 seul sens), elle contient un nombre d'éléments égale à *taille*,
- le premier élément est fictif (c'est un marqueur de début de liste sur lequel on positionne le curseur pour parcourir toute la liste, il ne fait pas réellement partie de la liste), il s'appelle *sentinel*,
- le dernier élément est le seul qui possède un suivant qui pointe sur *sentinel*,
- un nouvel élément ajouté à la liste s'insère toujours à la gauche d'un index compris entre 0 et *taille*, cet index étant tiré **aléatoirement** entre les 2 bornes ( *0 <= index <= taille* ),
- chaque élément de la liste possède une valeur *donnee* d'un type générique défini par *E*. La recherche d'un élément dans la liste se fait par comparaison avec cette valeur.

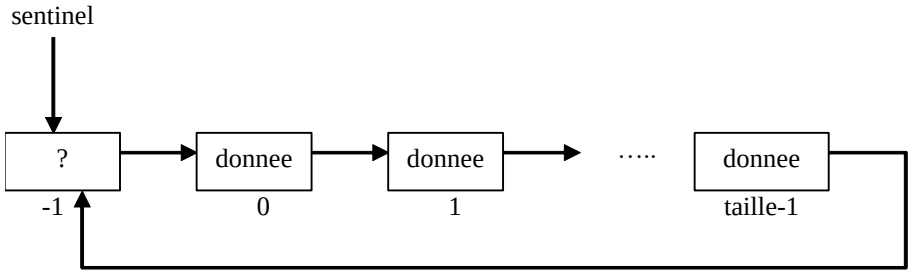


**Figure 1 : la liste simplement chaînée de la classe *Sac***

## *2. Intégration de la classe Sac dans l'API Java*

Dans l'*API Java* qui concerne les structures de données (paquetage *java.util*), les relations entre les classes doivent respecter une certaine organisation. Dans notre cas, si l'on veut que la classe *Sac* respecte les normes d'organisation imposées par l'API, **il faut que *Sac* hérite de *AbstractCollection***, *AbstractCollection* étant une classe abstraite qui implémente l'interface *Collection*.

Après consultation du code de la classe *AbstractCollection<E>*, on constate que :

- la plupart des méthodes sont déjà écrites (*toString()*, *isEmpty()*, *contains(Object o)*, *remove(Object o)* etc.). Il n'est donc plus nécessaire de les ré-écrire dans *Sac*,
- trois méthodes (ou presque) sont abstraites ce qui signifie que leur code doit être détaillé dans la classe *Sac* (qui hérite de cette classe abstraite). Il s'agit de *Iterator<E> iterator()*, *int size()* et *boolean add(E o)*.

### *3. Squelette de la classe Sac<E> (paquetage structdonnees)*

A. Attributs : *Element sentinel* et *int taille*.

B. Constructeur d'une liste vide : *Sac()*

création de *sentinel* (qui pointe sur elle-même) et *taille = 0*

C. Constructeur d'une liste à partir d'une *Collection<E> c* : *Sac ( Collection<E> c )*

1. créer une liste vide (en reprenant le code du constructeur d'une liste vide *Sac()*),
2. ajouter dans cette liste, en appelant la méthode *boolean add(E o)* de la classe *Sac*, chaque élément de la *Collection c* passée en paramètre.

Note : d'abord tester correctement la méthode *add(...)* avant de coder ce constructeur.

D. Une classe interne *Element* (voir ci-dessous).

E. Méthode *Iterator<E> iterator()* qui renvoie un nouvel objet de type *java.util.Iterator* (interface Java). Cet objet *Iterator*, renvoyé à la classe utilisatrice de la liste chaînée, va lui permettre (à la classe utilisatrice) d'effectuer 3 opérations sur cette liste :

1. *boolean hasNext ( )* : renvoie vrai si l'itérateur n'est pas en fin de liste.
2. *E next ( )* : déplace l'itérateur sur son voisin de droite et renvoie la donnée *E* du nouvel élément.
3. *void remove ( )* : supprime l'élément (type *Element*) désigné par l'itérateur.

! ! Attention : les 3 méthodes ci-dessus (*hasNext*, *next*, *remove*) sont des méthodes d'une classe interne à la classe *Sac* (voir plus loin) et ne sont donc PAS des méthodes de la classe *Sac*.

F. Méthode *int size()* qui renvoie la taille de la liste.

G. Méthode *boolean add( E o )* : crée un nouvel élément contenant la donnée "o" et l'ajoute dans la liste suivant le principe aléatoire expliqué ci-dessus. Elle renvoie faux si la taille (attribut *taille*) dépasse le maximum autorisé, c'est-à-dire *Integer.MAX_VALUE*.

### *4. La classe interne Element*

```
private class Element {

        // Attributs
        Element suiv ;          // Connexion à l'élément suivant de la liste
        E donnee ;              // Donnée stockée

        // Constructeur d'un élément de la liste
        Element ( E donnee, Element suiv ) { … }
        // Aucun accesseur et aucun modificateur à écrire !!
}
```

### *5. La classe interne Itr (qui implémente java.util.Iterator)*

Cette classe interne doit implémenter l'interface *java.util.Iterator* de l'API de Java. Cette implémentation de l'interface vous contraint à donner le code des 3 méthodes publiques de l'interface : *boolean hasNext ( )*, *E next ( )* et *void remove ( )*.

La classe *Itr* possède 2 curseurs (attributs de la classe) : *Element courant* et *Element precCourant*.

Pour l'opération *next()*, le curseur *courant* se déplace sur son voisin de droite et il renvoie la donnée *E* de l'élément sur lequel il pointe. Le deuxième curseur *precCourant* vient se placer juste à la gauche de *courant*.

Pour l'opération *remove()*, on supprime l'élément désigné par *courant*. Si l'opération est possible, *courant* vient se placer ensuite sur l'élément qui se trouve à sa gauche. L'API de Java vous oblige à gérer la suppression (*remove()*) de manière un peu particulière : il est interdit d'effectuer 2 (ou +sieurs) suppressions à la suite. En d'autres mots, 2 suppressions doivent nécessairement être entrecoupées d'un *next()*. C'est pour cette raison que la classe *Itr* possède 2 curseurs : *Element courant* et *Element precCourant*. Le rôle de ces 2 curseurs est d'empêcher une suppression 2 fois de suite sur la liste. Après une opération de type *next ( )* sur la liste, *precCourant* est toujours placé juste à la gauche de *courant*. Après une suppression par contre, on positionne *courant* et *precCourant* sur le même élément. Lors d'une nouvelle suppression, on teste d'abord l'égalité des références *courant* et *precCourant* : si il y a égalité, la suppression est interdite et l'utilisateur de l'instance de *Itr* devra d'abord passer par l'opération *next()*. Dans le cas contraire, l'opération *remove()* est possible.

Le constructeur de la classe *Itr* initialise simplement les attributs *courant* et *precCourant* en les faisant pointer tous les deux sur la sentinelle de la liste chaînée.

Note importante : à l'intérieur de la classe *Sac*, il n'est absolument pas nécessaire d'utiliser un objet *Iterator* pour parcourir la liste chaînée (son utilisation complique le codage inutilement).

### *6. Gestion d'itérateurs multiples sur une même liste*

Soit une classe extérieure *X* quelconque, utilisatrice d'une liste chaînée *Sac*. Rien n'interdit à cette classe de récupérer **plusieurs** itérateurs sur cette **même** liste par l'intermédiaire de la méthode *Iterator<E> iterator()* de la classe *Sac*. Dès lors, d'une part, par l'intermédiaire de ces itérateurs multiples, *X* pourra effectuer des opérations de type *hasNext()*, *next()* et *remove()* sur la liste. D'autre part, par l'intermédiaire des méthodes publiques de *Sac*, *X* pourra également effectuer des opérations de type *add(...)* et *size()* sur cette même liste.

Même si cela n'a pas été évoqué jusqu'à présent, cette possibilité pour une classe extérieure de gérer **plusieurs** itérateurs en simultané **sur une même liste** peut poser des problèmes d'incohérences dans les résultats des opérations effectuées par les itérateurs. Exemple : soit *it1* et *it2*, deux itérateurs distincts sur la même liste *theL*. On suppose qu'à un certain moment *it1* et *it2* pointent tous les deux (avec *courant* et *precCourant*) sur les mêmes éléments de la liste (pourquoi pas). Si par l'intermédiaire de *it1* on effectue l'opération *remove*, l'élément pointé par *courant* de *it1* disparaît de la liste (c'est le but de l'opération *remove*) ce qui rend *it2* **inutilisable** puisque le *courant* de *it2* pointe maintenant sur un élément qui ne fait plus partie de la liste (faites un dessin si vous n'êtes pas convaincu) ! !

Pour résoudre le problème, on va apporter un correctif au code des classes *Sac* et *Itr*, en adoptant la stratégie déjà mise en place par les classes de l'*API de Java* qui héritent du type *Collection*. Le principe est le suivant : la classe *Sac* va stocker un attribut supplémentaire dit « de synchronisation des itérateurs » de type entier appelé *comptModif* et initialisé à zéro. Pour une liste chaînée donnée, cet attribut sera une référence de comparaison unique pour un itérateur qui veut effectuer une opération sur la liste. Chaque itérateur possède également son propre attribut entier dit « de synchronisation avec la liste chaînée » appelé *comptModifAttendu* et initialisé à *comptModif* lors de la création de l'itérateur. On établit les règles suivantes pour empêcher un itérateur de faire une opération incohérente sur la liste :

- Il est **toujours** possible de créer un nouvel itérateur sur une liste (même si la liste est vide). Dès sa création on synchronise l'itérateur avec la liste en initialisant son attribut *comptModifAttendu* à la valeur de référence de la liste *comptModif*. Cet itérateur devient inutilisable dès qu'il essaye d'effectuer une opération de type *next*, *hasNext* ou *remove* sur la liste alors qu'il n'est plus synchronisé (c'est-à-dire : $comptModifAttendu \neq comptModif$).
- **Après avoir vérifié qu'il est synchronisé avec la liste**, un itérateur peut effectuer une opération de type *remove* sur celle-ci. On rend cet itérateur comme seul et unique itérateur valide sur la liste en

modifiant le compteur de référence (*comptModif*++) et en re-synchronisant l'attribut *comptModifAttendu* de cet itérateur avec la nouvelle valeur (*comptModifAttendu* = *comptModif*).

- Dès qu'un itérateur désire effectuer une opération de type *next* ou *hasNext*, on vérifie d'abord qu'il est synchronisé avec la liste (égalité stricte entre *comptModif* et *comptModifAttendu*). Si cet itérateur n'est pas synchronisé, une exception *java.util.ConcurrentModificationException* est lancée.
- L'opération *add(...)* de la classe *Sac* doit également prévoir (après insertion du nouvel élément) une modification du compteur de référence (*comptModif*++) pour empêcher tous les itérateurs existants d'effectuer une quelconque opération sur la liste par la suite.

Cette manière de procéder permet de donner un ordre de priorité entre les itérateurs de même qu'entre les opérations :

- Une opération de type *add* est toujours possible. Elle entraîne la désactivation de tous les itérateurs en cours.
- Un itérateur synchronisé peut effectuer sur la liste toutes les opérations autorisées pour un itérateur. Cas particulier : s'il effectue une opération de type *remove*, il devient **le seul** itérateur valide sur la liste.