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

R3.02 – TP1 – Liste chaînée

J-F. Kamp

Septembre 2023

Premier 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

TP1 : Liste chaînée

1. Objectifs

Construction de la classe *LinkedList.java* (paquetage *datastruct*) sous Eclipse qui implante une liste doublement chaînée avec sentinelle. On créera d'abord une interface *List.java* et ensuite *LinkedList.java* qui implémentera cette interface. L'usage de *JUnit4* et *Sonarlint* sont obligatoires.

2. Création d'un projet sous Eclipse

Après avoir lancé Eclipse (de préférence une version *Photon, Oxygen, Mars ou Neon*), suivre attentivement les étapes suivantes (laisser d'abord Eclipse choisir le *workspace* par défaut et fermer la page d'accueil *Welcome*) :

1. Sélectionner *File -> New -> Java Project* puis *Next*
2. Une boite de dialogue *Create a Java Project* s'ouvre :
   - Project name: *nomDuProjet* (*projLinkedList* par exemple)
   - Rubrique « JRE » : imposer *javaSE-11 (*ou *javaSE-1.8* si réseau enseignement*)*
   - Rubrique « Project layout » : sélectionner *Create separate folders for sources and class files*
   - *Next*
3. Nouvelle boite de dialogue *Java Settings* où dans la rubrique « Default output folder » il doit apparaître *nomDuProjet/bin*. Si c'est le cas, ne rien faire et cliquer sur *Finish*. Ensuite, une boite de dialogue apparait pour proposer la perspective *Java* de votre projet : cliquez sur « Yes ».

Dans votre répertoire *workspace* créé par Eclipse, vous devez constater l'existence de l'arborescence suivante : un répertoire *nomDuProjet* et puis 2 sous-répertoires *src* et *bin* dans lesquels on devra trouver respectivement les classes sources et les classes compilées.

3. L'interface List.java

Eventuellement sélectionner d'abord la perspective « Java » (en haut à droite dans Eclipse) si ce n'est déjà fait.

1. Créer en premier lieu le paquetage « datastruct » :
   - Sélectionner *File -> New -> Package*
   - Une boite de dialogue *New Java Package* s'ouvre : donner le nom du paquetage (*datastruct*) puis cliquer sur *Finish*
2. Créer l'interface *List.java* (paquetage *datastruct*) :
   - Sélectionner *File -> New -> Interface*
   - Une boite de dialogue *New Java Interface* s'ouvre : donner le nom de l'interface (*List*) puis cliquer sur *Finish*
3. Cette interface doit contenir les signatures de l'ensemble des méthodes *public* qui seront définies dans la classe *LinkedList.java* qui implémente cette interface. Voici la liste des méthodes de l'interface :
   - *public void insert (Object data) ;*

     Insertion d'un nouvel élément dans la liste. Il doit précéder l'élément *current* (à sa gauche). Le nouvel élément devient l'élément courant. Incrémenter également le nombre d'éléments de la liste.

- *public boolean delete () ;*

  Supprime l'élément courant. Repositionne l'élément courant sur son précédent (à sa gauche). Cas particulier : si le courant se trouve en tête de liste (le précédent est la sentinelle), repositionner l'élément courant sur son suivant (à sa droite). Décrémenter également le nombre d'éléments de la liste. Renvoie *faux* si la liste est vide.

- *public boolean contains (Object data) ;*

  Renvoie *vrai* si un élément de la liste contient la valeur *data*, *faux* dans le cas contraire.

- *public void add (int index, Object data) ;*

  Insertion d'un nouvel élément dans la liste à la position *index* (>= 0 et <= *size*). Le nouvel élément devient l'élément courant. Incrémenter également le nombre d'éléments de la liste.

- *public Object getValue () ;*

  Renvoie la valeur contenue dans l'élément courant.

- *public void setValue (Object newData) ;*

  Modifie le contenu de l'élément courant avec la nouvelle donnée passée en paramètre.

- *public boolean isEmpty () ;*

  Renvoie *vrai* si la liste ne contient aucun élément.

- *public int getSize () ;*

  Renvoie le nombre d'éléments de la liste.

4. La classe LinkedList.java

1. Créer la classe *LinkedList* (paquetage *datastruct*) qui doit implémenter *List* :
   - Sélectionner *File -> New -> Class*
   - Une boite de dialogue *New Java Class* s'ouvre :
     - préciser d'abord le paquetage *datastruct* si nécessaire (rubrique « Package »),
     - donner le nom de la classe (*LinkedList*),
     - ➔dans la rubrique « Interfaces », cliquez sur « Add », une nouvelle boite de dialogue *Implemented Interfaces Selection* s'ouvre. Dans la rubrique *Choose interfaces*, écrivez le nom de l'interface (c-à-d *List*) et sélectionnez dans la rubrique *Matching types* l'interface du paquetage *datastruct* et cliquez sur « Add » puis sur « OK ».
     - enfin cliquez sur *Finish*
2. La classe créée *LinkedList* aura 3 attributs (*sentinel*, *current*, *size*) et devra contenir la classe interne (voir ci-dessous) *Element*. Le squelette de la classe se présente alors comme suit :

```
class LinkedList implements List {
 // Attributs
 private Element sentinel ;    // pointe sur l'élément sentinelle
 private Element current ;     // pointe sur l'élément courant
 private int size ;            // nombre d'éléments dans la liste

 // Constructeur de liste vide (une sentinelle comme seul élément)
 public LinkedList () { … }
```

```
// Méthodes publiques (en + de celles de l'implémentation de l'interface List)
public void goToHead() { … }        // Placer le curseur sur la tête de liste
public void goToEnd() { … }         // Placer le curseur en fin de liste
public boolean next() { … }         // Placer le curseur sur son suivant si existe
public boolean previous() { … }     // Idem next() mais sur son précédent
public String toString() { … }      // Renvoie tout le contenu de la liste sous
                                    // forme textuelle (i.e. toutes les données)
public Object getValueAt(int index) // Renvoie la donnée à la position « index »
                                    // dans la liste

…
// La classe interne Element
private class Element {
      Element prev ;          // Connexion à l'élément précédent de la liste
      Element next ;          // Connexion à l'élément suivant de la liste
      Object theValue ;       // Donnée stockée

      // Constructeur d'un élément à insérer dans la liste doublement chaînée
      Element ( Element prev, Element next, Object data ) { … }
      // Pas de modificateurs ni accesseurs car attributs visibles par LinkedList
 }
}
```

5. Contrats

Vérifiez que votre code contient bien l'ensemble des contrats qui permettent de garantir que les classes développées assurent correctement les services qu'on leur attribue.

En pratique, 3 types de contrat doivent être écrits :

1. Pré-condition : condition à vérifier AVANT l'exécution d'une méthode publique. Consiste, en pratique en Java, à tester en tout début de méthode la validité des paramètres passés à celle-ci et à lancer une exception si la valeur de ces paramètres est incorrecte.
2. Post-condition : condition à vérifier APRES l'exécution d'une méthode publique ou privée pour contrôler l'accomplissement correct du travail demandé à la méthode. Consiste, en pratique en Java, à tester avec une (ou +sieurs) assertion(s) en fin de méthode, que l'objet modifié se trouve bien dans l'état demandé.
3. Invariant : condition à vérifier à tout moment pour s'assurer que l'objet reste dans un état cohérent « tout au long de sa vie ». Consiste, en pratique en Java, à tester avec une méthode privée *private boolean invariant()* la validité de chacun des attributs de l'objet. L'appel de la méthode *invariant()* doit se faire dès la création de l'objet et en fin d'exécution de presque chacune des méthodes.

6. La classe de test

Une classe de test *LinkedListTest* doit être créée séparément. On doit y trouver le test de l'ensemble des méthodes publiques de la classe *LinkedList*. Faire usage obligatoirement de *JUnit4*.

7. Théorie succincte des classes internes

Soit une classe *A* et une classe *InnA* interne à la classe *A*. La classe *InnA* est déclarée à l'intérieur de la classe *A* au même niveau que les méthodes et attributs de la classe *A*.

Exemple :

```
class A {
      // Attributs de A
      private int i, j ;
      private InnA var ; // InnA est la classe interne

      // Un constructeur pour A
      public A() {
            var = this.new InnA ( 23 ) ;  // (3)
            var.j = 2.5 ;                 // (1)
      }

      public InnA getInnA () { return var ; }

      private class InnA {
            // Attribut de InnA
            private double j ;

            public InnA ( int x ) {
                  this.j = x ;            // (2)
                  this.j = A.this.j ;     // (2)
            }
      }
}
```

Commentaires :

- La classe *A* compile, elle est classe englobante.
- *InnA* est une classe interne de *A*. Mais son véritable nom est *A.InnA* vis-à-vis des autres classes.
- Dans tous les cas et quelque soit la visibilité de *InnA* (même *private*), les champs privés des classes *A* & *InnA* sont accessibles depuis *A* et depuis *InnA* (voir (1) et (2)).
- La création d'un objet *InnA* passe forcément d'abord par la création d'un objet *A* (voir (3)).
- Dans la classe *InnA*, un attribut de *InnA* est accessible avec *this* tandis qu'un attribut de *A* est accessible avec *A.this* (voir (2)).

Soit une autre classe *B* du même paquetage que *A* :

- Vis-à-vis de cette classe *B*, la classe interne s'appelle *A.InnA*.
- Une variable de type *InnA* ne peut même pas être déclarée si *InnA* est privée (*private class InnA*).
- Dans tous les autres cas, il est possible de créer une instance de *InnA* mais que par l'intermédiaire d'une instance de *A* :

```
A var1 ;
A.InnA  var2 ;
var1 = new A() ;
var2 = var1.new InnA() ;
```

- La visibilité des champs de *InnA* n'a de sens que par rapport aux classes non englobantes qui utilisent des objets de type *InnA*. Dans ce cas :
  - *InnA* doit être de visibilité différente de *private* au minimum (sinon le type *InnA* est invisible),
  - Seuls les champs, de l'objet *InnA* créé, de visibilité *public*, *package* ou *protected* sont accessibles (règles habituelles d'accessibilité).