BUT informatique – R101-c

TP structures de données, algorithmes et programmation

Étienne ANDRÉ
Basé sur des documents de Vincent THOMAS

www.loria.science/andre/enseignement/R101c/

Version : 14 décembre 2021

Table des matières

Un peu de cours : les interfaces en Java **3**

1 Interface et type abstrait de données **14**

2 Implémentation contiguë **19**

3 Implémentation chaînée **23**

4 Implémentation d'une pile **30**

Les interfaces en Java

0.1 Structures de données et Java

0.1.1 Interface et TAD

La notion d'*interface* en Java correspond à celle de *type abstrait de données* vue en cours. Le cours de R101-c organisé en séparant les TAD (types abstraits de données **Liste**(E) par exemple) et leurs implémentations (liste chaînée, liste contiguë, ...) reprend la même distinction qu'entre classes et interfaces.

Ainsi :

— Un TAD correspond à une interface en Java. Un TAD et une interface permettent tous les deux de préciser les fonctions (pour un TAD) ou les méthodes (pour une interface) qui doivent exister mais sans préjuger de leur contenu.

— Une implémentation correspond à une classe Java qui implémente une interface. Par exemple, une liste chaînée sera représentée par une classe `ListeChainee` qui implémente l'interface `Liste`. Cela signifie que la classe `ListeChainee` devra posséder toutes les méthodes présentes dans l'interface `Liste`.

0.1.2 Algorithme logique et algorithme de programmation

La distinction entre algorithme logique et algorithme de programmation faite en cours va se retrouver en Java :

— un algorithme logique consiste à écrire un `main` ou des méthodes pour résoudre un problème algorithmique en utilisant les méthodes de l'interface sans préjuger de la manière dont elles fonctionnent;

— un algorithme de programmation consiste à écrire une classe qui implémente une interface. Cela consiste alors à écrire le code source des différentes méthodes de l'interface.

0.2 Exemple des Files

0.2.1 TAD et interface d'une File

0.2.1.1 TAD

En cours, nous avons vu le *Type abstrait de données (TAD)* **File**. Une file est caractérisée par les opérations suivantes :

fileVide	:		→ **File**(**Valeur**)
premier	:	**File**(**Valeur**) \ {fileVide()}	→ **Valeur**
estVideFile	:	**File**(**Valeur**)	→ **booléen**
adjfil	:	**File**(**Valeur**) × **Valeur**	→
supfil	:	**File**(**Valeur**) \ {fileVide()}	→

Le TAD d'une file permet de définir les opérations possibles sur les files sans savoir de quelle manière une file sera mise en œuvre.

0.2.1.2 Interface d'une file

Pour définir l'interface d'une file, chacune des fonctions du TAD doit être transformée en une méthode dans l'interface `File`.

— La fonction fileVide a pour objectif de construire une nouvelle file. Cette fonction sera remplacée par un *constructeur* en Java et ne sera donc pas déclarée dans l'interface `File`.

— Comme une méthode s'applique sur un objet, le premier paramètre de chaque fonction du TAD de type **File**(**Valeur**) correspond au paramètre caché `this` et n'apparaît donc pas dans la signature des méthodes de l'interface.

L'interface `File` s'écrira donc de la manière suivante :

```java
/**
 * interface qui represente une File de chaines independamment
 * de toute implementation
 *
 */
public interface File {

  /**
   * methode premier d'une file
   *
   * @return l'element en tete de file
   */
  public String premier();

  /**
   * methode qui permet de savoir si la file this est vide
   *
   * @return booleen qui vaut vrai si et seulement si la file ne
   *         contient aucun element
   */
  public boolean estVideFile();

  /**
   * methode qui ajoute un element a la file this
   *
   * @param s
   *            element a ajouter en queue de la file
   */
  public void adjfil(String s);

  /**
   * methode qui supprime l'element en tête de la file
   */
  public void supfil();

}
```

Cette interface peut aussi être représentée par le diagramme de classe de la figure 1. L'interface est représentée par une boite et chaque méthode par un élément de cette boite (en précisant les types des paramètres et le type de retour).

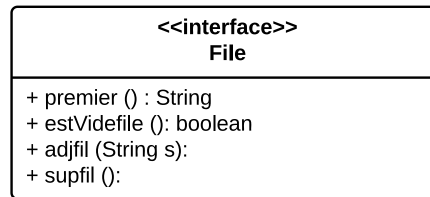


FIGURE 1 – Diagramme de classe représentant l'interface `File`. Le symbole '+' devant les méthodes signifie que ces méthodes sont publiques.

0.2.2 Algorithme logique et programme principal

0.2.2.1 Algorithme logique

Un *algorithme logique* résout un problème en utilisant uniquement le TAD fourni. Voici par exemple l'algorithme affichant l'ensemble des éléments de la file et en les retirant progressivement.

```
1 Algorithme logique
2 fonction afficherEtSupprimerÉlémentsFile(f InOut : File(chaîne))
3 début
4     tant que non estVideFile(f) faire
5         valeur ← premier(f)
6         écrire(valeur)
7         supfil(f)
8     fin tq
9 fin
```

0.2.2.2 Programme principal

L'algorithme logique se traduira en Java par une méthode qui *utilisera* l'interface `File`.

```java
/**
 * classe qui résout le probleme consistant a afficher le contenu d'une
    file et a en supprimer le contenu
 */
public class AfficheFile {

  /**
  * methode qui affiche une File
  * @param f file a afficher
  */
  public void afficherEtSupprimerElementsFile(File f)
  {
    // tant qu'il reste des elements
    while (!f.estVideFile()) {
      // recuperer la valeur et l'afficher
      String valeur = f.premier();
      System.out.println(valeur);
      // supprimer la valeur de la file
      f.supfil();
    }
  }
}
```

```java
/**
* classe principale qui utilise l'algorithme logique
```

```
3  */
4  public class Prog
5  {
6    /**
7     * programme principal
8     */
9    public static void main(String[] args) {
10     File f;
11     // creation de la file a faire
12     AfficheFile fileAfficheur = new AfficheFile();
13     fileAfficheur.afficherEtSupprimerElementsFile(f);
14   }
15 }
```

0.2.3 Implémentation contiguë

0.2.3.1 Algorithmes de programmation

Une file (voir figure 2) peut être représentée par un type composite constitué :

1. d'un tableau de chaînes de type :

$$\textbf{tableau chaîne}[1..MAXTAB]$$

2. d'un entier *tête* désignant le début de la file ;
3. d'un entier *nb* désignant le nombre d'éléments de la file.

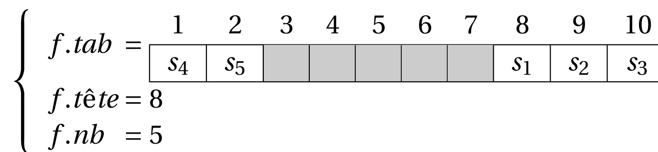


FIGURE 2 – Représentation contiguë d'une file (rappel du cours). Elle représente une file de skieurs avec 5 éléments (*MAXTAB* vaut 10).

À partir de cette représentation, il est possible d'écrire les **algorithmes de programmation** des différentes fonctions du TAD (voir polycopié de cours).

Fonction fileVide La fonction fileVide() a pour objectif d'initialiser une file. C'est une file qui ne possède aucun élément ($f.nb$ vaut 0) et dont la tête pointe en début de tableau ($f.tête$ vaut 1).

```
1 fonction fileVide() : File(Valeur)
2 début
3     f.nb ← 0
      /* pour éviter le cas particulier de l'adjonction dans une file
         vide                                                        */
4     f.tête ← 1
      /* (en programmation, il faudra aussi créer/initialiser le
         tableau f.tab)                                              */
5     retourner f
6 fin
```

Fonction premier La fonction premier(f) a pour objectif de retourner le premier élément de la file. La fonction retourne simplement la valeur située dans la case désignée par la tête de la file.

```
1 fonction premier(f : File(Valeur)) : Valeur
2 début
3 |  retourner f.tab[f.tête]
4 fin
```

Fonction estVideFile La fonction estVideFile(f) a pour objectif de retourner un booléen valant vrai si et seulement si la file est vide. Comme la file est vide si et seulement son nombre d'éléments est 0, la fonction retourne le booléen $f.nb = 0$.

```
1 fonction estVideFile(f : File(Valeur)) : booléen
2 début
3 |  retourner (f.nb = 0)
4 fin
```

Fonction adjfil La fonction adjfil(f, v) a pour objectif d'ajouter la valeur v à la file f. La valeur s'ajoute en queue de file (car premier a été pris en tête). On calcule donc la case d'ajout de la nouvelle valeur (en prenant en compte le fait qu'on puisse dépasser du tableau), on y affecte cette valeur et on augmente le nombre d'éléments de la file.

```
1 fonction adjfil(f InOut : File(Valeur), v : Valeur)
2 début
  |  /* Recherche de l'indice d'ajout : attention à la fin du tableau
  |     */
3 |  indice ← ((f.tête + f.nb − 1) mod MAXTAB) + 1
  |  /* Alternative: si f.tête + f.nb > MAXTAB alors
  |     indice ← f.tête + f.nb − MAXTAB sinon indice ← f.tête + f.nb      */
4 |  f.tab[indice] ← v
5 |  f.nb ← f.nb + 1
6 fin
```

Fonction supfil La fonction supfil(f) a pour objectif de supprimer l'élément en tête de file. Il suffit simplement de décaler la tête d'un cran (en prenant en compte la sortie du tableau) et de diminuer le nombre d'éléments de la file. La valeur supprimée peut rester dans le tableau (pas de suppression explicite nécessaire), car elle n'est plus accessible ; elle sera écrasée lors de la suite de la manipulation de la file.

```
1 fonction supfil(f InOut : File(Valeur))
2 début
3 |  f.tête ← (f.tête mod MAXTAB) + 1
  |  /* Alternative: si f.tête = MAXTAB alors f.tête ← 1 sinon
  |     f.tête ← f.tête + 1                                               */
4 |  f.nb ← f.nb − 1
5 fin
```

0.2.3.2 Classe FileContigue

En se basant sur les algorithmes ci-dessus, il est possible de définir la classe `FileContigue`.

— Les attributs de la classe `FileContigue` correspondront à la représentation de la file. La classe `FileContigue` possédera donc trois attributs :

1. un tableau de chaînes `tab`,
2. un entier `tete` désignant l'indice de début de la file,
3. un entier `nb` désignant le nombre d'éléments dans la file.

L'entier `MAXTAB` désignant le nombre maximal d'éléments stockables dans cette file n'est pas utile puisqu'il correspond à `tab.length`.

— Le constructeur de la classe `FileContigue` remplacera la fonction fileVide() et prendra en paramètre la taille du tableau qui stockera la file.

— Les autres méthodes de la classe `FileContigue` proviendront de l'interface `File` et devront être écrites conformément aux algorithmes de programmation. Le paramètre « *f* : **File**(**Valeur**) » des fonctions issues du cours sera remplacé par l'objet `this` sur lequel la méthode est appelée.

```
1  /**
2   * implementation d'une file contigue
3   * la classe FileContigue implemente l'interface File
4   */
5  public class FileContigue implements File {
6
7    /**
8     * la tete de la file
9     */
10   private int tete;
11
12   /**
13    * le nombre d'elements de la file
14    */
15   private int nb;
16
17   /**
18    * le tableau des elements contenus dans la file
19    * MAXTAB correspond a la taille du tableau tab
20    */
21   private String[] tab;
22
23   /**
24    * constructeur vide correspond a fileVide
25    *
26    * @param maxTaille correspond a la taille maximale de la File
27    */
28   public FileContigue(int maxTaille) {
29     if (maxTaille < 1)
30       maxTaille = 1;
31     // creation du tableau de String
32     this.tab = new String[maxTaille];
33     // initialisation de tete et nb
34     this.tete = 1;
35     this.nb = 0;
36   }
37
38   /**
39    * methode premier correspond a la fonction premier
40    *
```

```
41     * @return  le premier element de la file
42     */
43   public String premier() {
44     return (this.tab[this.tete]);
45   }
46
47   /**
48    * permet de savoir si une file est vide
49    *
50    * @return true si et seulement si la file est vide
51    */
52   public boolean estVideFile() {
53     return this.nb == 0;
54   }
55
56   /**
57    * permet d'ajouter un element dans la file
58    *
59    * @param s chaine a ajouter en queue de file
60    */
61   public void adjfil(String s) {
62     // cherche la case où stocker la valeur
63     int numCase = this.nb + this.tete;
64     if (numCase > this.tab.length)
65       numCase = numCase - this.tab.length;
66     // stocke la valeur
67     this.tab[numCase] = s;
68     // augmente la taille de la file
69     this.nb++;
70   }
71
72   /**
73    * supprime la tete de la file
74    */
75   public void supfil() {
76     // augmente la tete
77     this.tete = this.tete + 1;
78     if (this.tete > this.tab.length)
79       this.tete = this.tete - this.tab.length;
80     //diminue la taille
81     this.nb--;
82   }
83
84 }
```

0.2.3.3 Diagramme UML

La relation entre l'interface `File` et la classe `FileContigue` est ce qu'on appelle une relation d'*implémentation*.

Cette relation s'écrit en Java grâce au mot clef `implements`, et est représentée de manière graphique dans le diagramme de classe de la figure 3. La relation d'implémentation apparaît comme une flèche en pointillé de la classe `FileContigue` correspondant à l'implémentation vers l'interface `File`.

0.2.3.4 Utilisation de la classe FileContigue

Pour utiliser la file contiguë dans l'algorithme logique, il suffit de remplacer la ligne de création mise en commentaire dans le programme précédent par un appel au constructeur de la classe `FileContigue` et des adjonctions.

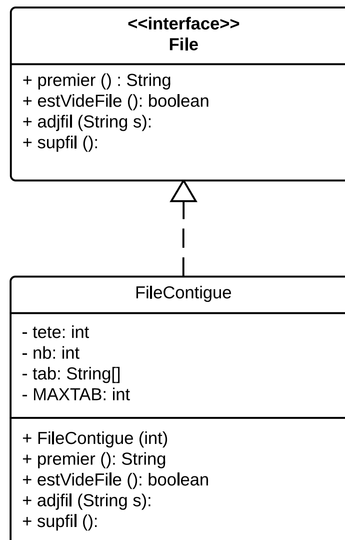


FIGURE 3 – Diagramme de classe représentant la relation d'implémentation entre l'interface `File` et la classe `FileContigue`

```java
/**
 * classe qui utilise un algorithme logique permettant d'afficher
 * le contenu d'une file
 */
public class Prog {

  /**
   * programme principal
   *
   * @param args inutilise
   */
  public static void main(String[] args) {
    //creation d'une file geree de maniere contigue
    File f = new FileContigue(10);

    //remplissage de la file
    f.adjfil("Adeline");
    f.adjfil("Bertrand");
    f.adjfil("Celine");
    f.adjfil("Delphine");

    //appel de la methode qui contient l'algorithme logique
    AfficheFile fileAfficheur = new AfficheFile();
    fileAfficheur.afficherEtSupprimerElementsFile(f);
  }

}
```

0.2.4 Implémentation chaînée

0.2.4.1 Classe `FileChainee`

Il est possible de proposer une autre implémentation de l'interface `File` à l'aide d'une autre mécanique interne (par exemple le chaînage d'éléments comme vu dans une liste représentée de manière chaînée).

On peut alors définir une classe `FileChainee` qui implémente elle aussi l'interface `File`,

mais dont les attributs privés et le contenu des méthodes — c'est-à-dire les algorithmes de programmation — sont différents.

0.2.4.2 Représentation graphique

Les deux implémentations `FileContigue` et `FileChainee` implémentent alors la même interface `File`. Ces deux implémentations coexistent dans le même programme : chaque implémentation se trouve dans son fichier respectif (`Filecontigue.java` et `FileChainee.java`), et représente une manière différente de manipuler des files. C'est ensuite à la personne qui programme de choisir la manière qui lui parait la plus adaptée.

Ces différentes implémentations apparaissent dans le même diagramme de classe, comme représenté sur la figure 4.

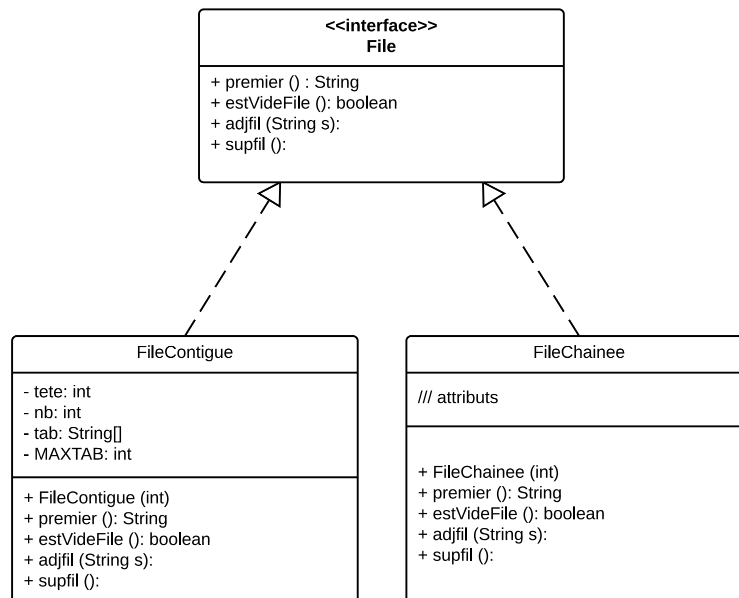


FIGURE 4 – Diagramme de classe représentant l'interface `File` et les deux implémentations `FileContigue` et `FileChainee`

0.2.4.3 Utilisation de la classe `FileChainee`

Pour utiliser la file chaînée dans l'algorithme logique, il suffit de remplacer la ligne `File f = new FileContigue(10)` par un appel au constructeur de `FileChainee`.

```
1  /**
2   * classe qui utilise un algorithme logique permettant d'afficher
3   * le contenu d'une file
4   */
5  public class Prog {
6
7    /**
8     * programme principal
9     * on utilise desormais une FileChainee
10    *
11    * @param args inutilise
12    */
13   public static void main(String[] args) {
14     //creation d'une file de maniere chainee
15     File f = new FileChainee(10);
16
```

```
    //remplissage de la file
    f.adjfil("Adeline");
    f.adjfil("Bertrand");
    f.adjfil("Celine");
    f.adjfil("Delphine");

    //appel de la methode qui contient l'algorithme logique
    AfficheFile fileAfficheur = new AfficheFile();
    fileAfficheur.afficherEtSupprimerElementsFile(f);
  }

}
```

0.2.5 Choix d'implémentation

Ainsi, en choisissant le constructeur utilisé (dans le cas des files `new FileChainee()` ou `new FileContigue(10)`), il est possible de sélectionner une implémentation plutôt qu'une autre.

```
import java.util.Scanner;

/**
 * classe qui choisit l'implementation utilisee et affiche le contenu
 * de la File
 */
public class Prog {
  /**
   * programme principal
   */
  public static void main(String[] args) {
    // demande utilisateur
    Scanner sc = new Scanner(System.in);
    System.out.println("Choisir implementation");
    System.out.println("(1) contigue (2) chainee");
    int choix = sc.nextInt();

    // creation d'une file en fonction de choix
    File f = null;
    if (choix == 1)
      f = new FileContigue(10);
    else
      f = new FileChainee(10);

    // remplissage de la file
    f.adjfil("Adeline");
    f.adjfil("Bertrand");

    // appel de la methode qui contient l'algorithme logique
    AfficheFile fileAfficheur = new AfficheFile();
    fileAfficheur.afficherEtSupprimerElementsFile(f);
  }
}
```

Le choix de l'implémentation peut avoir de l'importance en fonction du programme utilisé et des opérations effectuées. Par exemple, dans le cas des listes, une liste chaînée permet de gérer rapidement les insertions dans la liste, mais prend du temps lorsqu'on souhaite accéder à un élément de la liste (puisqu'il faut parcourir la liste depuis la tête). Inversement, une liste contiguë permet d'accéder rapidement à n'importe quel élément de la liste mais prend du temps lorsqu'on souhaite y faire une insertion (puisqu'il faut alors décaler les anciennes valeurs pour laisser de la place pour la nouvelle valeur).

Ces comparaisons seront l'objet de la SAÉ 1.02 « comparaisons d'approches algorithmiques ».

TP 1

Interface et type abstrait de données

Objectifs

Au cours du TP, vous apprendrez

1. à définir un type abstrait de données sous Java grâce à la notion d'interface ;
2. à utiliser une interface pour écrire un algorithme logique ;
3. à réinvestir vos connaissances pour écrire des programmes Java.

À rendre

À l'issue de ce TP, vous devez rendre sur Arche une archive (au format `.zip` exclusivement) contenant (uniquement) les fichiers sources suivants :

— l'interface `Liste.java` ;

— la classe `ProgLogiqueSimple.java` complétée.

(Aucun autre fichier ne devra être rendu.)

Attention, dans tous les TP, l'archive devra être nommée « `S1G_nom.zip` », où « `G` » est votre numéro de groupe, et « `nom` » votre nom (sans espace).

1.1 Liste de sous-titres

L'objectif des trois premiers TPs va être de manipuler une liste de sous-titres et de pouvoir trier les sous-titres dans l'ordre chronologique.

La classe `SousTitre` vous est fournie sur Arche. Cette classe possède

1. deux attributs :
   (a) un attribut `temps` de type `int` représentant la date du sous-titre
   (b) un attribut `phrase` de type `String` représentant le texte associé au sous-titre
2. plusieurs méthodes :
   (a) un constructeur avec deux paramètres de type `int` et `String` correspondant aux attributs de l'objet à créer ;
   (b) une méthode `String toString()` chargée de l'affichage d'un sous-titre ;
   (c) une méthode `int compareTo(SousTitre st)` chargée de comparer le sous-titre courant au sous-titre `st` passé en paramètre ;
   (d) des accesseurs `int getTemps()` et `String getPhrase()`.

On prendra comme exemple la liste de sous-titres suivante (non triée pour le moment) :

264	"Batman: Vite, à la Batmobile!!"
98	"Robin: Batman, j'entends un rire de dément"
255	"Robin: Nom d'un petitbonhomme, mais c'est bien sûr"
258	"Batman: Oui encore un coup du Joker"
152	"Batman: Mais de qui peut-il s'agir ?"

1.2 Interface Liste(SousTitre)

La notion d'interface en Java permet de spécifier des méthodes sans préciser comment ces méthodes seront mises en œuvre. Cette notion permet ainsi de définir les types abstraits de données (TAD) vus dans le cours (voir chapitre 1 du polycopié).

Pour rappel, le TAD **Liste**(**Valeur**) possède les opérations suivantes :

Opération		Domaine		Résultat
tête	:	**Liste**(**Valeur**)	→	**Place**
val	:	**Liste**(**Valeur**) × (**Place** \ {nil})	→	**Valeur**
suc	:	**Liste**(**Valeur**) × (**Place** \ {nil})	→	**Place**
finliste	:	**Liste**(**Valeur**) × **Place**	→	**booléen**
lisvide	:		→	**Liste**(**Valeur**)
adjtlis	:	**Liste**(**Valeur**) × **Valeur**	→	
adjlis	:	(**Liste**(**Valeur**) \ {lisvide()}) × (**Place** \ {nil}) × **Valeur**	→	
suplis	:	(**Liste**(**Valeur**) \ {lisvide()}) × (**Place** \ {nil})	→	
suptlis	:	**Liste**(**Valeur**) \ {lisvide()}	→	
adjqlis	:	**Liste**(**Valeur**) × **Valeur**	→	
supqlis	:	**Liste**(**Valeur**) \ {lisvide()}	→	
chglis	:	(**Liste**(**Valeur**) \ {lisvide()}) × (**Place** \ {nil}) × **Valeur**	→	

Pour simplifier l'écriture des classes et des programmes Java,

— on ne s'intéressera qu'aux premières opérations ci-dessus (de tête à suplis inclus) ;

— on se concentrera sur des listes d'objets de type `SousTitre` ;

— on supposera que les places dans la liste sont représentées par des entiers (ce qui n'est pas toujours vrai — cf. TP 4 concernant l'implémentation des piles).

1.2.1 Transformation des fonctions en méthodes

Le fait de transformer les fonctions vues dans le module de R101c en des méthodes vues dans le module de *programmation1* impliquera de faire disparaître certains paramètres des fonctions.

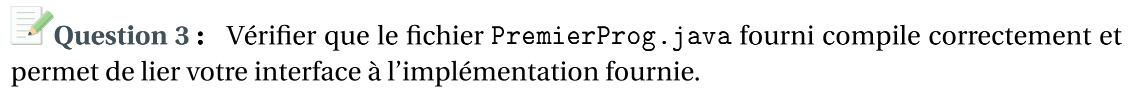
Question 1 : Pour chaque opération du TAD **Liste**, réfléchir à la manière dont l'opération apparaîtra dans l'interface `Liste` (constructeur ou méthode et paramètres).

1.2.2 Déclaration interface Liste

Question 2 : Déclarer correctement l'interface `Liste` en fonction des résultats de la question précédente.

1.2.3 Validation de l'interface

On vous fournit sur l'ENT le fichier `ListeProf.class` correspondant à une première implémentation.



Question 3 : Vérifier que le fichier `PremierProg.java` fourni compile correctement et permet de lier votre interface à l'implémentation fournie.

Si la compilation du programme `PremierProg.java` génère des erreurs, c'est que votre déclaration de l'interface n'est pas conforme à ce qui est attendu. Faites les modifications qui s'imposent avant de passer à la suite.

À l'issue de cette partie, les classes obtenues sont structurées selon le diagramme de classe de la figure 1.1.

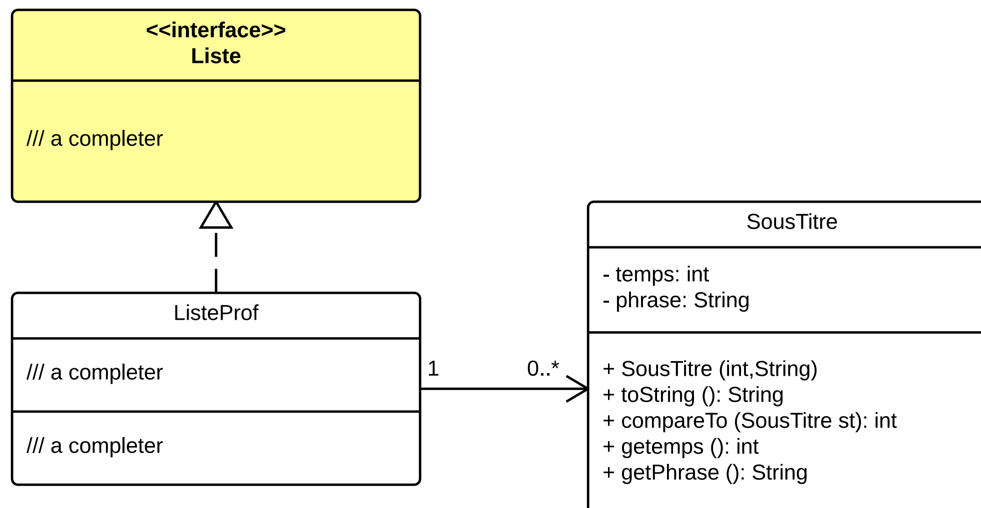


FIGURE 1.1 – Diagramme de classe qui présente l'interface `Liste` et les classes `SousTitre` et `ListeProf`. La flèche pleine entre `ListeProf` et `SousTitre` signifie qu'un objet `ListeProf` possède des objets `SousTitre` en attribut. En jaune, l'interface `Liste` à écrire.

1.2.4 Javadoc

Question 4 : Ajouter la javadoc à l'interface `Liste`.

1.3 Algorithme logique

1.3.1 Ajout et affichage

Le début de la classe `ProgLogiqueSimple` vous est fourni sur arche :

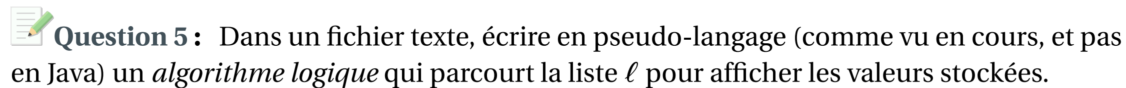
```
/**
 * premier programme a ecrire par les etudiants
 */
public class ProgLogiqueSimple {

  public static void main(String []args)
  {
    Liste l=new ListeProf();
    l.adjtlis(new SousTitre(264, "Batman: Vite a la batmobile"));
    l.adjtlis(new SousTitre(98, "Robin: Batman, j'entends un rire
   dement"));
    l.adjtlis(new SousTitre(255, "Robin: Nom d'un petit bonhomme"));
    l.adjtlis(new SousTitre(258, "Batman: Oui, encore un coup du
   Joker"));
```

```
    l.adjtlis(new SousTitre(152, "Batman: Mais de qui peut il s'agir"));

    // **** A COMPLETER PAR LES ETUDIANTS ****
    // ** afficher le contenu de la liste **
    throw new Error("A COMPLETER");
  }
}
```

Cette classe est à compléter pour afficher le contenu de la liste en la parcourant. Vous noterez que le début du programme utilise l'interface `Liste` pour déclarer la variable `l` et cacher le fait qu'il s'agit d'une liste de la classe `ListeProf`.



Question 5 : Dans un fichier texte, écrire en pseudo-langage (comme vu en cours, et pas en Java) un *algorithme logique* qui parcourt la liste ℓ pour afficher les valeurs stockées.

Faire valider par votre enseignant.

Question 6 : Recopier cet algorithme en commentaire dans la méthode `main` de la classe `ProgLogiqueSimple.java`.

Traduire l'algorithme logique précédent en Java en utilisant l'interface `Liste`.

1.3.2 Algorithme de tri (optionnel)

On souhaite trier les sous-titres par date pour les afficher dans l'ordre. La méthode de tri est déclarée dans la classe `Trier` qui vous est fournie. Cette méthode prend en paramètre un tableau de sous-titres et retourne en résultat la liste triée.

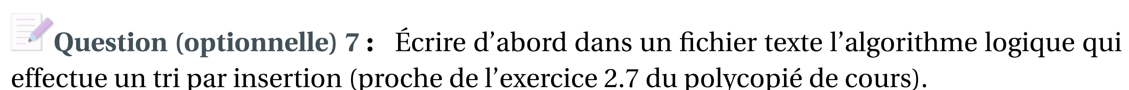
```
/**
 * classe en charge de trier une liste
 */
public class Trier {

  /**
   * methode de tri par insertion
   *
   * @param tab
   *            le tableau des sous titres a trier
   * @return la liste triee
   */
  public Liste trier(SousTitre[] tab) {
    // **** A COMPLETER PAR LES ETUDIANTS ****
    throw new Error("A COMPLETER");

  }
}
```

1.3.2.1 Tri par insertion

Un tri par insertion consiste à considérer chaque sous-titre du tableau l'un après l'autre et de l'insérer au bon endroit dans une nouvelle liste en respectant l'ordre du tri (voir figure 1.2).



Question (optionnelle) 7 : Écrire d'abord dans un fichier texte l'algorithme logique qui effectue un tri par insertion (proche de l'exercice 2.7 du polycopié de cours).

Question (optionnelle) 8 : Traduire l'algorithme logique en Java en utilisant les méthodes de l'interface `Liste`.

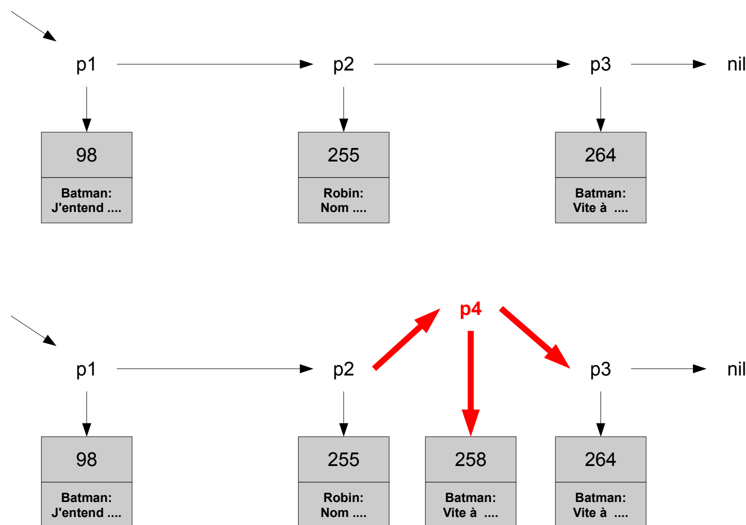
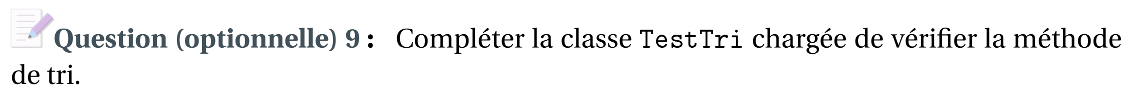


FIGURE 1.2 – Exemple de l'insertion du 4e élément en maintenant la liste triée

1.3.2.2 Test de tri



Question (optionnelle) 9 : Compléter la classe `TestTri` chargée de vérifier la méthode de tri.

Prochaine séance

Le TP de la prochaine séance abordera les listes contiguës. Elles sont à réviser (chapitre 2.2 du polycopié du cours).

TP 2

Implémentation contiguë

Objectifs

Ce TP a pour objectifs de

1. savoir mettre en œuvre une implémentation continue de liste ;
2. savoir implémenter une interface.

À rendre

À l'issue de ce TP, vous devez rendre sur Arche une archive (au format `.zip` exclusivement) contenant le fichier source suivant :

— la classe `ListeContigue.java` complétée.

Rappel : dans tous les TP, l'archive devra être nommée « `S1G_nom.zip` », où « `G` » est votre numéro de groupe, et « `nom` » votre nom (sans espace).

2.1 Liste contiguës

Maintenant que l'algorithme logique a été écrit, nous allons implémenter l'interface liste grâce aux algorithmes de programmation vus dans le chapitre 2.2 du polycopié de cours. L'objectif de ce TP est d'écrire une classe `ListeContigue` qui implémente l'interface `Liste`.

2.1.1 Rappel

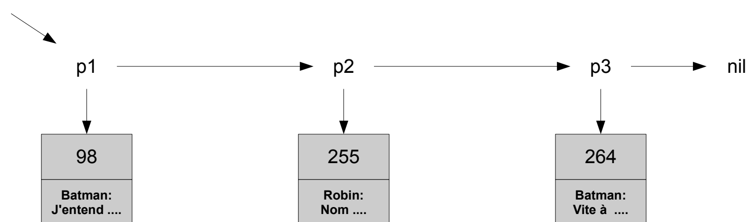


FIGURE 2.1 – Liste abstraite

Une liste contiguë est représentée en interne sous la forme d'un tableau de valeurs. Les éléments sont les uns derrière les autres dans l'ordre des indices du tableau [1].

L'implémentation contiguë d'une liste est définie comme le produit cartésien constitué

1. Relire le polycopié de cours (chapitre 2.2) pour plus d'informations.

1. d'un tableau de valeurs permettant de stocker les valeurs de la liste ;
2. du nombre d'éléments de la liste (pour connaître le nombre de valeurs effectivement stockées dans la liste).

2.1.2 Attributs

Ce produit cartésien s'exprimera en Java sous la forme d'attributs de la classe `ListeContigue` :

1. `tab` le tableau de sous-titres, `tab.length` désigne la taille de ce tableau ;
2. `nbElements` le nombre (variable) de cases de ce tableau utilisées pour gérer la liste.

Le constructeur de la classe `ListeContigue` prend un paramètre entier `tailleMax` permettant de moduler la taille de `tab` et donc la mémoire nécessaire à la liste.

La classe à construire est représentée graphiquement dans le diagramme de classe de la figure 2.2 (en jaune la classe à écrire).

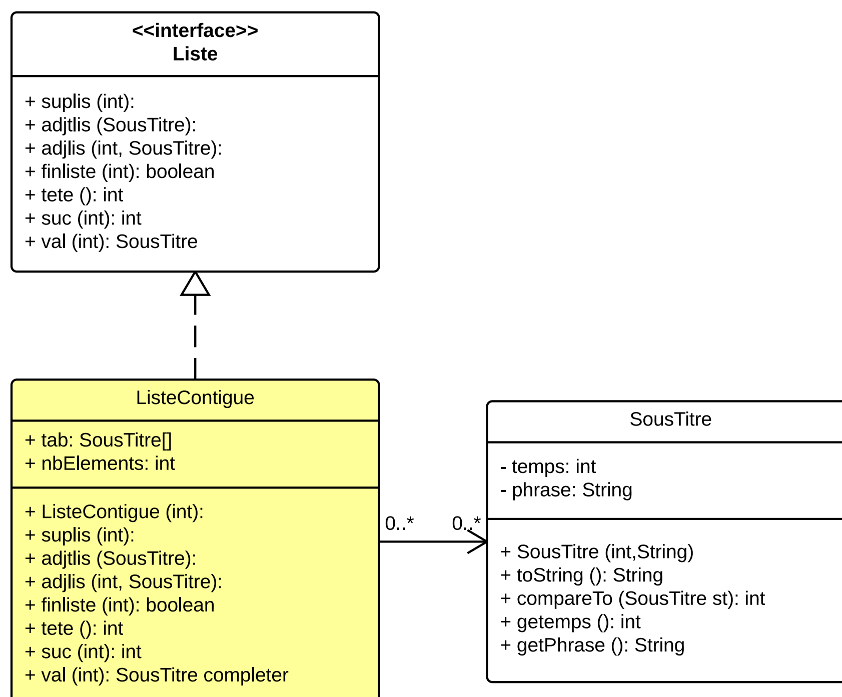


FIGURE 2.2 – Diagramme de classe qui présente la classe `ListeContigue` à écrire. Le lien (association) vers la classe `SousTitre` signifie que la classe `ListeContigue` possède plusieurs attributs de type `SousTitre`.

Le code suivant doit construire la liste de la figure 2.1. Cette liste sera représentée en mémoire en Java par le schéma-mémoire de la figure 2.3

```
1 Liste l = new ListeContigue(9);
2 l.ajtlis(new SousTitre(264,"Batman: Vite ..."));
3 l.ajtlis(new SousTitre(255,"Robin: Nom ..."));
4 l.ajtlis(new SousTitre( 98,"Batman: J'entends ..."));
```

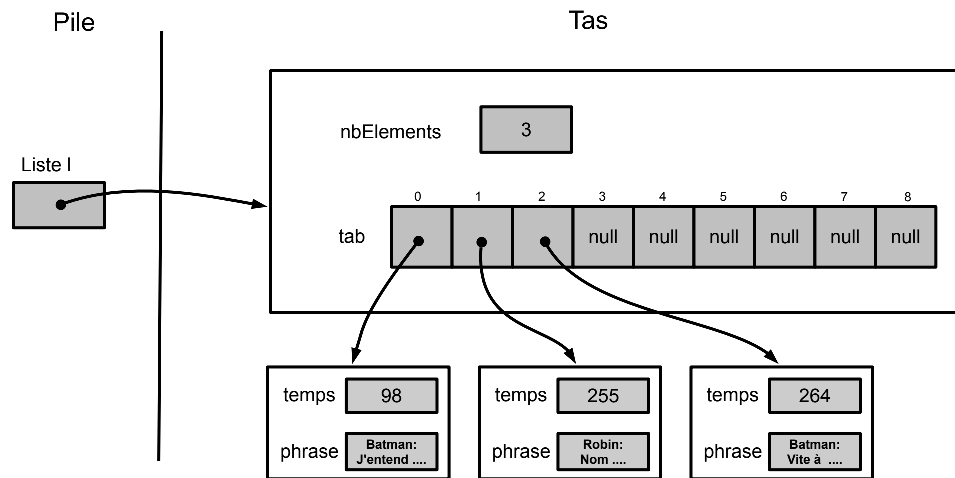


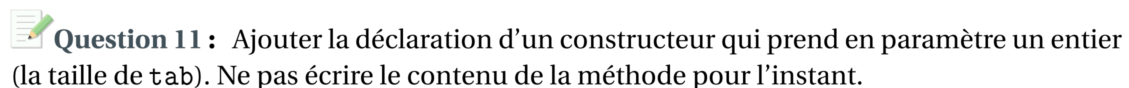
FIGURE 2.3 – Implémentation contiguë en Java de la liste précédente

2.2 Démarche incrémentale

2.2.1 Déclaration des méthodes

Comme vous avez déclaré certaines méthodes dans l'interface `Liste`, il est nécessaire que la classe `ListeContigue` possède ces méthodes.

Question 10 : Recopier chacun des profils des méthodes de l'interface dans la classe.



Question 11 : Ajouter la déclaration d'un constructeur qui prend en paramètre un entier (la taille de `tab`). Ne pas écrire le contenu de la méthode pour l'instant.

Pour que votre classe puisse compiler même si le contenu des méthodes n'est pas écrit, on peut ajouter l'instruction suivante dans chacune des méthodes à écrire :

```
1 throw new Error("A completer");
```

Cette instruction génère une erreur qui stoppe le programme dés que Java exécute cette ligne.

Par exemple, cela donne pour la méthode `tete()` le code suivant :

```
1 /**
2  * retourne la premiere place de la liste
3  * @return tete de liste
4  */
5 public int tete()
6 {
7   throw new Error("A COMPLETER");
8 }
```

Question 12 : Compléter les méthodes en ajoutant les instructions levant des `Error`.

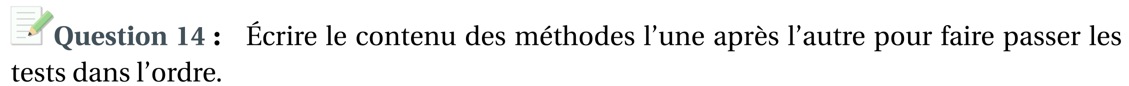
2.2.2 Lancement des tests

Une fois la classe complétée, la compilation devrait se passer correctement mais les tests devraient tous échouer puisque les méthodes génèrent des échecs.

Question 13 : Compiler la classe et lancer les tests pour vérifier qu'ils se lancent bien mais qu'ils échouent tous.

2.2.3 Écriture du corps des méthodes

Après chaque méthode et avant de passer à la suivante, vérifier que les tests liés à cette méthode sont valides. Bien prendre le temps de lire les messages d'erreur de l'application de test pour savoir quelle est l'origine de l'erreur (cela peut être une autre méthode à écrire).

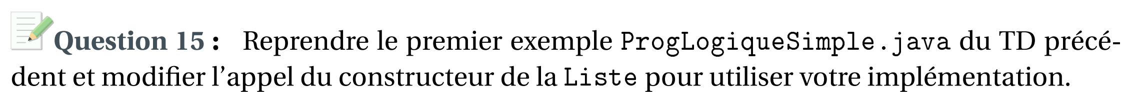


Question 14 : Écrire le contenu des méthodes l'une après l'autre pour faire passer les tests dans l'ordre.

Pour chaque méthode, vous traiterez un exemple avec un dessin, écrirez l'algorithme en commentaire dans le source Java, puis, vous ajouterez le code Java entre ces commentaires.

2.3 Utilisation de l'implémentation

2.3.1 Exemple simple



Question 15 : Reprendre le premier exemple `ProgLogiqueSimple.java` du TD précédent et modifier l'appel du constructeur de la `Liste` pour utiliser votre implémentation.

Vérifier que l'application fonctionne correctement.

2.3.2 Tri par insertion (optionnel)

Question (optionnelle) 16 :
Faire de même pour la classe `Trier` et vérifier que le résultat est correct.

Prochaine séance

Le prochain TP traitera des listes chaînées. Le chapitre 2.3 du polycopié de cours est à réviser pour la prochaine séance.

TP 3

Implémentation chaînée

Objectifs

Ce TP a pour objectifs de

1. savoir mettre en œuvre une implémentation chaînée de liste ;
2. masquer la construction d'objet avec une Factory et une interface.

À rendre

À l'issue de ce TP, vous devez rendre sur Arche une archive (au format .zip exclusivement) contenant (uniquement) le fichier source suivant :

— la classe ListeChainee.java

3.1 Présentation

La partie précédente se concentrait sur l'implémentation contiguë d'une liste ; cette partie va se concentrer sur une implémentation chaînée conformément au cours (voir chapitre 2.3 du polycopié de cours). Ce TP consiste à écrire une classe ListeChainee implémentant l'interface Liste.

À la fin de ce TP, vous disposerez donc de trois types de listes différentes :

1. la classe ListeProf (fournie sous forme de .class) ;
2. la classe ListeContigue du TP 2 ;
3. la classe ListeChainee de ce TP.

Ces différentes listes implémentent toutes l'interface Liste et sont donc interchangeables (elles réalisent le même objectif).

3.2 Liste chaînée

3.2.1 Rappels

Une implémentation chaînée d'une liste consiste à représenter explicitement les successeurs dans la structure de données.

Désormais, les éléments qui se suivent dans la liste ne se suivent pas forcément dans la représentation physique. Dans le cours, les valeurs sont stockées dans un tableau dont les cases sont de type <val: SousTitre, suc:int>. La place où se trouve l'élément suivant

dans la liste est donnée par la valeur du champ *suc*[1]. Chaque élément connaît donc l'élément suivant avec lequel il est chaîné.

Comme les éléments ne sont plus contigus les uns aux autres, il faut pouvoir facilement référencer les emplacements libres pour ajouter un élément ou pour libérer la place réservée lors d'une suppression. Plusieurs approches sont envisageables mais nous allons simplement représenter les places libres comme les places ayant −2 pour successeur[2].

En cours,

— la liste est représentée par un tableau tab de [-1..MAXTAB] de couples <val: SousTitre, suc:entier>;

— la tête de la liste de travail est stockée dans le tableau tab à l'indice 0;

On supposera que nil, la fin de la liste, est représenté par l'entier -1.

3.2.2 Attributs et constructeur

Un choix de modélisation un peu différent a été fait en Java. Plutôt que de stocker la valeur de la tête de la liste à l'indice 0, on a préféré lui dédier un attribut spécifique.

Ainsi, l'attribut tete désigne la place à laquelle trouver le premier élément de la liste.

Le diagramme de classe de la figure 3.1 décrit les liens que possède la classe ListeChainee avec le reste de l'application.

3.2.2.1 Classe MaillonEntier

Afin de représenter le produit cartésien <val: SousTitre, suc:entier>, la classe MaillonEntier vous est fournie. Elle permet de stocker, au sein du même objet, la valeur de la liste ainsi que la place de l'élément suivant.

La classe MaillonEntier possède

— un attribut val de type SousTitre correspondant au sous-titre stocké;

— un attribut suc de type int correspondant à l'indice où trouver le prochain sous-titre;

— des méthodes get et set pour accéder et modifier les attributs.

3.2.2.2 Classe ListeChainee

Le produit cartésien ListeChainee s'exprimera en Java sous la forme d'attributs de la classe ListeChainee

— tete l'indice de tête de la liste;

— tab le tableau de MaillonEntier, chaque maillon stockant une valeur de sous-titre et le l'indice du maillon suivant dans tab.

Le constructeur de la classe ListeChainee prendra un paramètre entier permettant de moduler la taille de tab, donc la mémoire nécessaire à la liste. À la construction, il faut penser à initialiser une liste dont toutes les places sont libres (successeur à −2) conformément à la figure 3.2.

1. Pour plus d'information, se référer au chapitre 2.3 du polycopié de cours.
2. pour d'autres approches, voir polycopié de cours

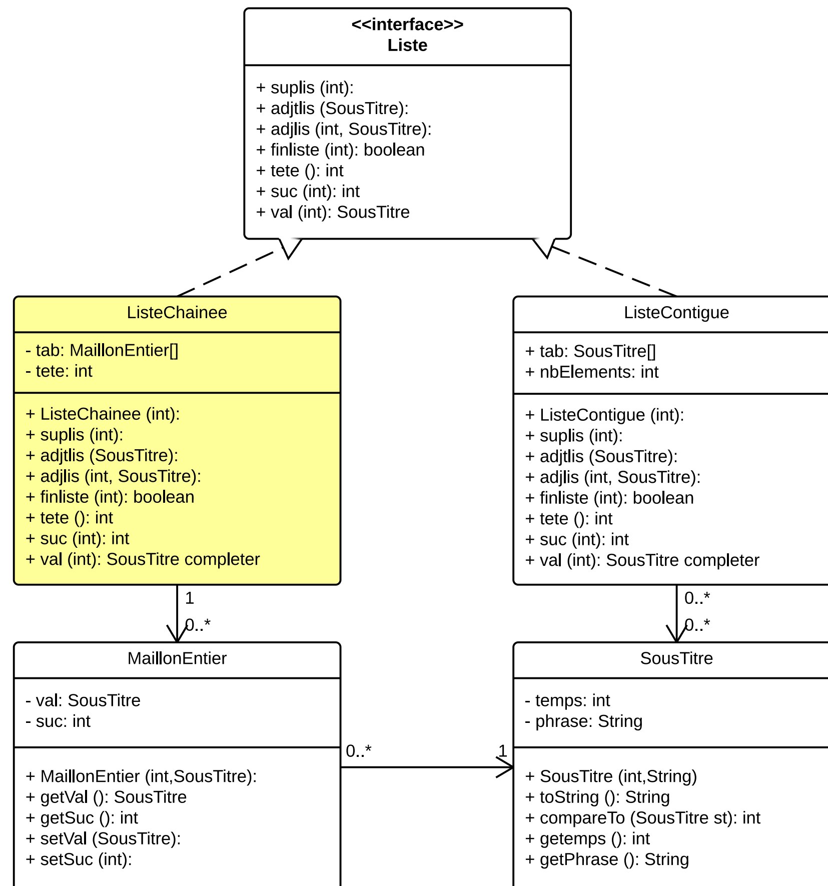


FIGURE 3.1 – La classe `ListeChainee` à écrire implémente l'interface `Liste`. Elle possède en attribut un tableau d'objets de type `MaillonEntier`. Chaque `MaillonEntier` possède un objet `SousTitre` en attribut. La classe `ListeContigue` reste présente.

3.2.3 Gestion de la place libre

Afin de gérer les places libres, deux méthodes à usage interne (donc *privées*) sont à ajouter à la classe `ListeChainee`

- une méthode `int retournerPlaceLibre()` qui retourne la première place libre;
- une méthode `void libererPlace(int p)` qui transforme la place `p` passée en paramètre en place libre.

Vous **penserez à écrire puis à utiliser ces fonctions** pour les méthodes d'ajout et de suppression dans la liste.

3.2.4 Exemple d'utilisation

La suite d'instructions suivante doit conduire au schéma mémoire de la figure 3.3

```
Liste l = new ListeChainee(7);
l.ajtlis(new SousTitre(264,"Batman: Vite ..."));
l.ajtlis(new SousTitre(255,"Robin: Nom ..."));
l.ajtlis(new SousTitre( 98,"Batman: J'entends ..."));
```

- l'attribut `tete` valant 2, la liste commence à la place 2 et le premier élément correspond au `SousTitre` ayant pour `temps` 98;
- la place de l'élément suivant est indiquée par l'attribut `suc` du maillon situé à l'indice 2. Cette place est 1;

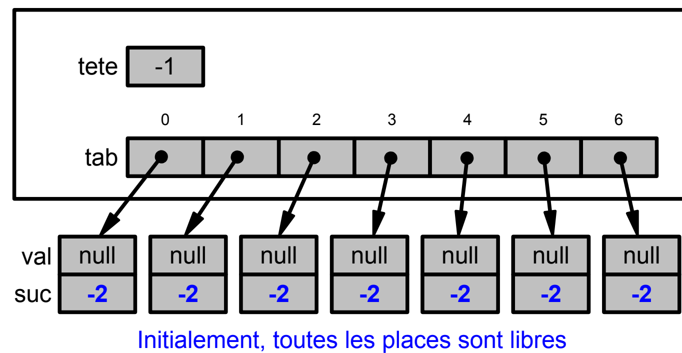


FIGURE 3.2 – Initialisation d'un objet de type `ListeChainee`

— le second élément est donc situé à la place 1 et correspond au `SousTitre` ayant pour `temps` 255;

— la place de l'élément suivant est indiquée par l'attribut `suc` à l'indice 1, à savoir 0;

— le troisième élément est donc à la place 0 et correspond au `SousTitre` ayant pour `temps` 264;

— comme la place de l'élément suivant est -1, la liste ne possède plus d'autres éléments.

De plus, s'il fallait ajouter un élément, l'ajout se ferait à la première place libre, indiquée par un attribut successeur ayant pour valeur -2, à savoir la place 3.

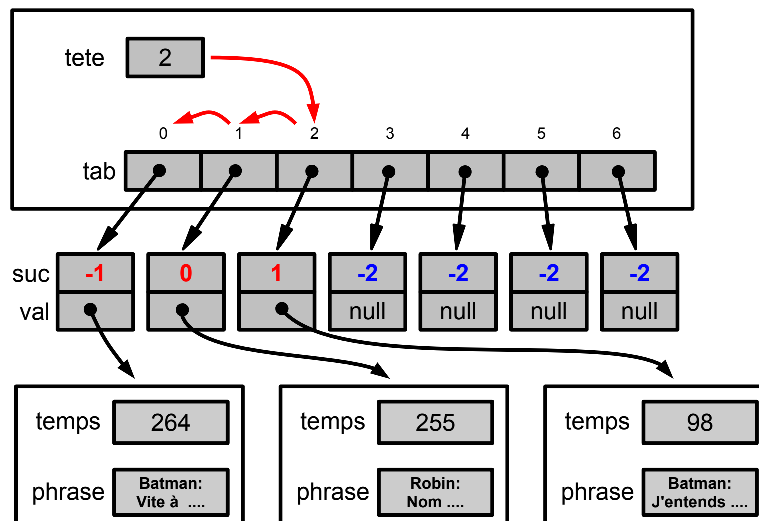


FIGURE 3.3 – Résultat des instruction du programme. En rouge la représentation de la liste de travail, en bleu, les places libres.

3.3 Démarche incrémentale

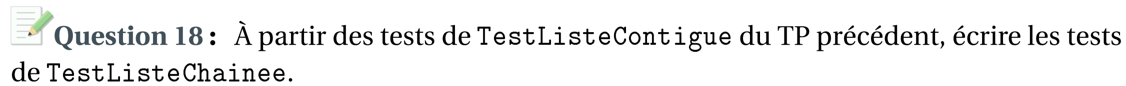
3.3.1 Déclaration des méthodes

Comme vous avez déclaré certaines méthodes dans l'interface `Liste`, il est nécessaire que la classe `ListeChainee` possède ces méthodes.

Question 17 : Recopier chacun des profils des méthodes de l'interface dans la classe.

3.3.2 Écriture des tests

Le squelette de la classe de test `TestListeChainee` est donné parmi les classes fournies dans Arche. Faites attention à ne pas écraser le `main` de cette classe.



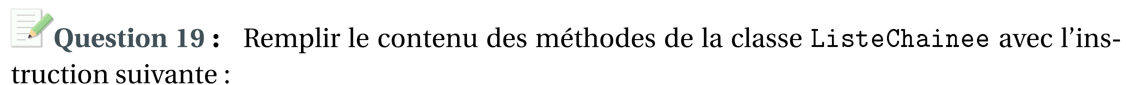
Question 18 : À partir des tests de `TestListeContigue` du TP précédent, écrire les tests de `TestListeChainee`.

3.3.3 Lancement des tests

Pour pouvoir compiler et lancer les tests, il est nécessaire de compléter les méthodes de la classe `ListeChainee` (en particulier les méthodes qui nécessitent une valeur de retour). Comme le TP précédent, vous remplirez ces méthodes avec l'instruction suivante qui génère une erreur lorsque Java l'exécute.

```
throw new Error("A COMPLETER");
```

Cela permet de ne pas avoir à compléter ces méthodes pour le moment mais uniquement lorsqu'elles sont utiles pour passer un test.



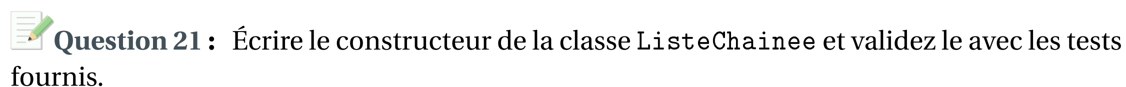
Question 19 : Remplir le contenu des méthodes de la classe `ListeChainee` avec l'instruction suivante :

```
throw new Error("A COMPLETER");
```

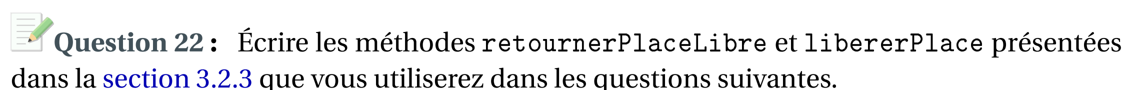
La compilation ne devrait pas générer d'erreur, mais les tests devraient échouer puisque l'appel de chaque méthode génère une erreur.

Question 20 : Compiler les classes et vérifier que vous pouvez lancer les tests.

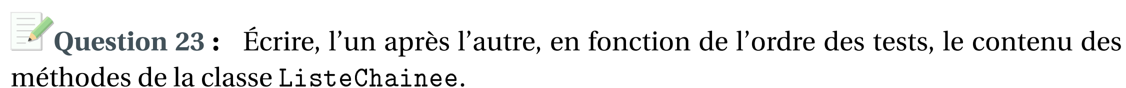
3.3.4 Écriture du corps des méthodes



Question 21 : Écrire le constructeur de la classe `ListeChainee` et validez le avec les tests fournis.



Question 22 : Écrire les méthodes `retournerPlaceLibre` et `libererPlace` présentées dans la section 3.2.3 que vous utiliserez dans les questions suivantes.



Question 23 : Écrire, l'un après l'autre, en fonction de l'ordre des tests, le contenu des méthodes de la classe `ListeChainee`.

Pour chaque méthode, commencez par traiter un exemple de test sur un dessin, écrivez l'algorithme en commentaire dans le source Java, puis, ajoutez le code Java correspondant entre ces commentaires.

Après chaque méthode et avant de passer à la suivante, vérifier que les tests liés à cette méthode sont valides.

3.4 Utilisation de la classe `ListeChainee`

3.4.1 Exemple simple

Question 24 : Reprendre la classe `ProgLogiqueSimple` du TD précédent et modifier l'appel du constructeur de la `Liste` pour utiliser votre implémentation.

Question 25 : Vérifier que l'application fonctionne correctement.

3.4.2 Tri par insertion (optionnel)

Question (optionnelle) 26 : Faire de même pour la classe `Trier` et vérifier que le résultat est correct.

3.5 Usine de Liste - Factory (optionnel)

3.5.1 Création de la Factory

Les trois classes `ListeProf`, `ListeContigue` et `ListeChainee` implémentent l'interface `Liste` et sont donc interchangeables (voir programme ci dessous).

```
String choix;
//modification de choix
Liste l;
switch(choix)
{
  case "prof":
    l = new ListeProf();
    break;
  case "contigue":
    l = new ListeContigue(100);
    break;
  case "chainee":
    l = new ListeChaine(100);
    break;
}

//suite des operations identique independemment de choix
l.adjtlis(new SousTitre(10, "test"));
```

On souhaite récupérer une implémentation de liste à partir d'une classe `DistributeurListe` chargée de créer des implémentations de listes en fonction de la demande. La classe `DistributeurListe` possède une seule méthode `fournirListe` qui prend en paramètre une chaîne de caractères et retourne comme résultat une implémentation de `Liste` dont le type correspond à la chaîne de caractère passée en paramètre comme le fait le programme ci-dessus.

Question (optionnelle) 27 : Compléter la classe `DistributeurListe` fournie.

3.5.2 Test de rapidité

Maintenant que les Listes sont construites de la même manière grâce à l'usine, il est possible de mener des tests pour comparer les temps d'exécution. La classe `EvaluerTemps`

contient le programme ci-dessous qui évalue le temps nécessaire à chaque liste pour faire les opérations.

Dans ces circonstances **et pour ces opérations**, la liste chaînée va 5 fois plus vite que la `listeContigue` (et le rapport de performance va augmenter en fin des tailles des listes manipulées).

```
/**
 * programme pour evaluer le temps necessaire a chaque liste
 * pour faire des operations complexes
 */
public class EvaluerTemps {

  public static void main(String [] args)
  {
    //les types des listes testees
    String[] types={"prof","contigue","chainee"};

    //createur de liste
    DistributeurListe distributeur=new DistributeurListe();
    SousTitre t=new SousTitre(10, "Test");

    //lancement des tests
    for (int numTest=0;numTest<types.length;numTest++)
    {
      //on demande la creation de la liste a tester
      String nom=types[numTest];
      Liste liste=distributeur.fournirListe(nom);

      //sauvegarde le temps avant les tests
      long debut=System.nanoTime();
      System.out.println("*****************\nLance les tests de "+nom);

      //les tests
      //on fait num fois
      int num=10000;
      for (int j=1;j<num;j++)
      {
        int k=99;
        //k ajouts puis k suppressions
        for (int i=0;i<k;i++)
          liste.adjtlis(t);
        for (int i=0;i<k;i++)
          liste.suplis(liste.tete());
      }

      //sauvegarde le temps apres les test
      long fin=System.nanoTime();
      System.out.println("fin des tests de "+nom);

      //performance difference de temps
      System.out.print("temps passe: ");
      System.out.println(((fin-debut)/num)+" nanosecondes par boucle");
      System.out.println("*****************\n");
    }
  }
}
```

TP 4

Implémentation d'une pile

Objectifs

Ce TP a pour objectifs

1. savoir mettre en œuvre une pile en Java à l'aide références ;
2. manipuler les références Java.

De manière concrète, dans ce TP, vous allez devoir

- déclarer une interface `Pile` contenant le descriptif des opérations possibles sur les piles ;
- implémenter une `Pile` en utilisant les références Java (classe `PileReference`) ;
- écrire un algorithme logique utilisant la classe `PileReference`.

À rendre

À l'issue de ce TP, vous devez rendre sur Arche une archive (au format `.zip` exclusivement) contenant (uniquement) les fichiers sources suivants :

- l'interface `Pile.java` complétée ;
- la classe `PileReference.java` ;
- la classe `Postfixe.java` avec la méthode `evaluer`.

4.1 TAD Pile et interface

4.1.1 Rappel

Une pile est une liste dans laquelle les adjonctions et suppressions se font à une seule extrémité appelée sommet [1].

Pour rappel, les seules opérations autorisées sur les piles sont les opérations suivantes :

pileVide	:		→ **Pile**(**Valeur**)
sommet	:	**Pile**(**Valeur**) \ {pileVide()}	→ **Valeur**
estVidePile	:	**Pile**(**Valeur**)	→ **booléen**
empiler	:	**Pile**(**Valeur**) × **Valeur**	→
dépiler	:	**Pile**(**Valeur**) \ {pileVide()}	→

L'algorithme logique qui sera implanté sera celui de l'évaluation d'une expression postfixée (voir exercice 2.14 du polycopié de cours).

Nous nous limiterons donc à des Pile(entier).

1. pour plus d'information, se référer au polycopié de cours

4.1.2 Déclaration de l'interface

Question 28 : Déclarer en Java l'interface `Pile` avec les opérations qui s'imposent.

4.2 Implémentation d'une pile à l'aide de références Java

4.2.1 Principe général

On souhaite implémenter une pile en utilisant les références Java[2]. Le principe général s'applique à toute structure linéaire comme les listes et est proposé dans le polycopié de cours.

Pour cela, vous allez devoir implémenter deux classes

1. la classe `PileReference` a pour objectif de représenter la pile, elle implémente l'interface `Pile`. La classe `PileReference` possédera un seul attribut `tete` de type `Maillon`.
2. la classe `Maillon` est une classe intermédiaire qui a pour objectif de chaîner les éléments contenus dans la pile. Cette classe `Maillon` possédera deux attributs.
    - un attribut `valeur` de type `int` chargé de stocker la valeur du maillon courant ;
    - un attribut `suivant` de type `Maillon` chargé de stocker la référence vers le maillon suivant.

La classe `Maillon` vous est fournie sur Arche.

```java
/**
 * permet de gerer un maillon de la pile
 * une maillon possede une valeur et un lien vers le maillon suivant
 */
public class Maillon {

  /**
   * la valeur stockee dans le maillon
   */
  public int valeur;

  /**
   * le lien vers le Maillon suivant
   */
  public Maillon suivant;

  /**
   * constructeur de Maillon
   * @param element l'element stocke (de type entier)
   * @param suiv le lien vers le maillon suivant
   */
  public Maillon(int element,Maillon suiv)
  {
    this.valeur=element;
    this.suivant=suiv;
  }
}
```

L'idée est la même que le TP précédent sauf que la classe `Maillon` a remplacé l'attribut `suc` de type `int` par un attribut de type `Maillon`. Cela revient à remplacer le tableau précédent qui stockait les valeurs directement par le tas (voir figure 4.1).

2. Cela se construit de manière analogue avec les pointeurs que vous verrez en C

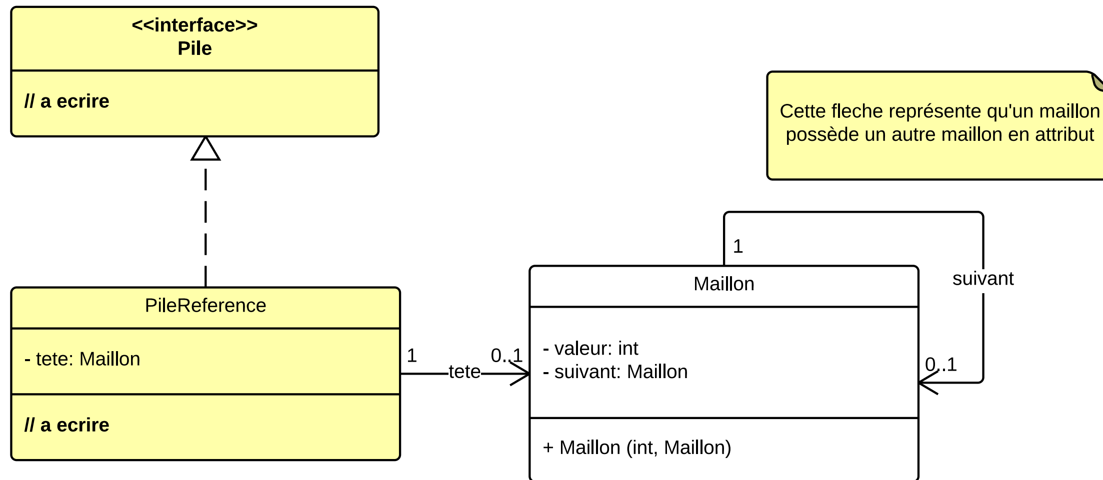


FIGURE 4.1 – Diagramme de classe représentant les liens entre l'interface `Pile`, la classe `PileReference` et la classe `Maillon`. Les classes à écrire sont représentées en jaune.

Une pile vide correspond simplement à un objet `Pile` pour lequel le maillon de tête est égal à `null`. Ensuite, chaque ajout dans la pile nécessite de créer un nouveau `Maillon` en initialisant correctement son attribut `suivant` et à le chaîner en tête des maillons déjà existants.

4.2.2 Exemple de schéma mémoire

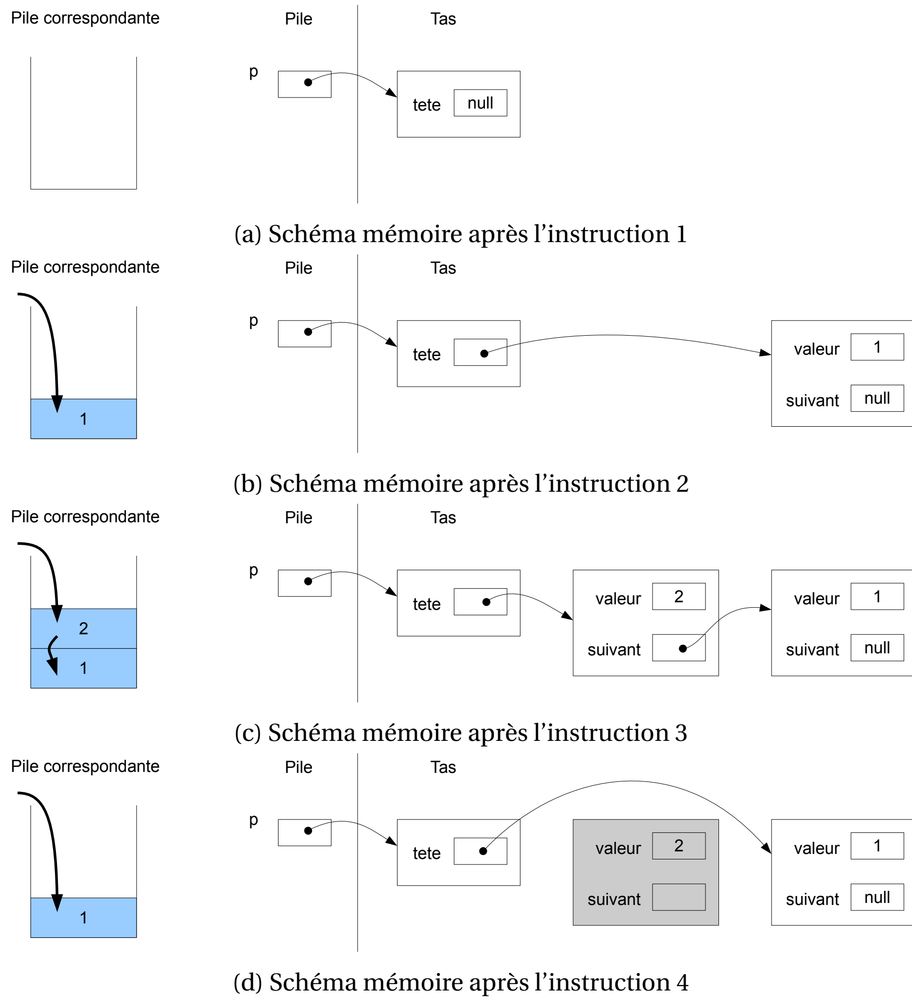
La figure 4.2 décrit la manière dont la `Pile` fonctionne sur l'exécution du programme suivant :

```java
/**
 * un programme simple avec une Pile
 */
public class ProgPile {

  /**
   * programme principal
   * @param args param inutilise
   */
  public static void main(String[] args) {
    //instruction 1
    Pile p=new PileReference();
    //instruction 2
    p.empiler(1);
    //instruction 3
    p.empiler(2);
    //instruction 4
    p.depiler();
    int a=p.sommet();
    //a vaut 1
    System.out.println(a);
  }
}
```

4.2.3 Mise en œuvre

Question 29 : Écrire le squelette de la classe `PileReference` qui implémente l'interface `Pile`.



(a) Schéma mémoire après l'instruction 1

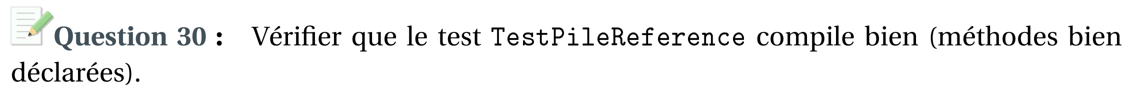
(b) Schéma mémoire après l'instruction 2

(c) Schéma mémoire après l'instruction 3

(d) Schéma mémoire après l'instruction 4

FIGURE 4.2 – Évolution du schéma mémoire

Vous penserez à compléter ce squelette avec un constructeur vide et des méthodes contenant l'instruction `throw new Error("a completer");`.



Question 30 : Vérifier que le test `TestPileReference` compile bien (méthodes bien déclarées).

4.2.4 Algorithmes de programmation

Question 31 : Écrire le contenu des méthodes de la classe `PileReference` de manière incrémentale en faisant passer successivement les tests de la classe `TestPileReference`.

4.3 Algorithme logique

4.3.1 Évaluation postfixée

On souhaite évaluer une expression postfixée grâce à la méthode `evaluer` dans la classe `Postfixe`[3].

- Cette méthode prend en paramètre un tableau de `String` correspondant à l'expression supposée correcte en notation postfixée. Chacun de ces `String` correspond à un opérateur ou un nombre.
- Cette méthode retourne comme résultat un `int` correspondant à l'évaluation de l'expression passée en paramètre.
- Cette méthode utilise en interne une `PileReference` pour effectuer ce calcul conformément à l'exercice 2.14 du polycopié de cours.
- Si vous êtes sous Java7, vous pouvez utiliser un `switch` basé sur un objet de type `String`.

Remarque 1

Pour convertir un `String` en entier, utilisez la méthode `Integer.parseInt(s)`[a].
Par exemple

```
1 String s="45";
2 int a=Integer.parseInt(s);
```

a. Cet appel est différent des appels de méthode que vous avez l'habitude de faire. La méthode est appelée à partir de la classe et pas d'un objet — on parle alors de méthode *statique*. Le fonctionnement sera détaillé dans un futur module.

Question 32 : Écrire la méthode `evaluer` dans la classe `Postfixe` fournie sur Arche.

Le programme suivant devrait retourner pour résultat 48.

```
1 /**
2  * algorithme logique qui utilise des Piles
3  */
4 public class AlgoLogique {
5
6   public static void main(String[] args)
7   {
```

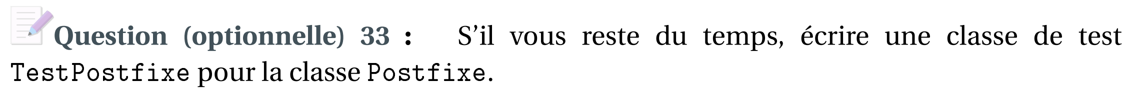
3. Vous ferez attention à ce que la classe `Postfixe` s'écrit avec un seul e

```
    //probleme notation polonaise inversee
    String[] valeurs={"12","38","+","2","-","."};

    Postfixe postfixe=new Postfixe();
    int resultat=postfixe.evaluer( valeurs);
    System.out.println("le resultat doit etre 48: "+resultat);
  }
}
```

4.3.2 Test de la méthode `evaluer` (optionnel)

Votre méthode `evaluer` sera testée automatiquement. Vous pouvez tenter d'améliorer cette méthode en écrivant vos propres tests et en les faisant passer.



Question (optionnelle) 33 : S'il vous reste du temps, écrire une classe de test `TestPostfixe` pour la classe `Postfixe`.