La classe `ArrayList`

Maria-Virginia Aponte, François Barthélémy

NFA031

1 Introduction : Les limites des collections de taille fixe

En programmation, on a souvent besoin de gérer une collection de données de même type : la liste des produits commandés par un internaute, l'ensemble de cartes d'un jeu de cartes, la file des clients en attente dans un guichet, ou encore les résultats d'un sondage. Dans de nombreux cas, la taille de cette collection peut varier tout au long du programme. Par exemple, la liste des références des produits commandés augmente à chaque produit ajouté au panier, et diminue quand un produit est retiré de la commande.

Quelle structure de données Java utiliser pour modéliser ce genre de collection ? Une première approche consiste à utiliser un tableau pour stocker les éléments :

```
String[] commandeReferences;
```

Cette approche présente plusieurs inconvénients majeurs. Tout d'abord, il faut déterminer à l'avance la taille maximale du tableau, ce qui n'est pas toujours possible ou souhaitable. Si on surestime cette taille, on gaspille de la mémoire ; si on la sous-estime, le programme ne pourra pas fonctionner correctement.

Ensuite, même en connaissant une taille maximale raisonnable, la gestion des ajouts et suppressions devient complexe. Sachant que la taille d'un tableau ne peut plus varier après avoir été créé, pour ajouter un nouveau produit à une commande déjà pleine, il faudrait :

1. créer un nouveau tableau, plus grand ;
2. copier les éléments de l'ancien tableau dans le nouveau ;
3. ajouter le nouvel élément ;
4. faire pointer la variable d'origine vers le nouveau tableau.

Soit en gros le code suivant :

```
String nouveauProduit = .........;
String[] tmp = new String[commandeReferences.length + 1];
for (int i = 0; i < commandeReferences.length; i++) {
   tmp[i] = commandeReferences[i];
}
// ajout du nouveau produit dans la dernière case
tmp[tmp.length - 1] = nouveauProduit;
// la variable qui désigne la commande doit pointer sur le
// nouveau tableau :
commandeReferences = tmp;
```

Pour supprimer un élément, un code similaire sera également nécessaire. Cette approche est lourde à mettre en place et coûteuse : en espace, car pour chaque élément ajouté on doit potentiellement créer un nouveau tableau, et en temps, car on doit alors recopier tous les éléments du tableau existant dans le nouveau tableau.

Heureusement, Java fournit dans ses bibliothèques plusieurs classes permettant de modéliser de manière simple et efficace les collections automatiquement redimensionnables. Ces structures de stockage ont des éléments de même type, et leur taille varie automatiquement à chaque ajout ou retrait d'élément, sans intervention du programmeur.

2 Présentation de la classe ArrayList

Un ArrayList permet de stocker une suite d'éléments de même type, avec un fonctionnement proche de celui des tableaux, mais avec la particularité d'être automatiquement redimensionnable.

2.1 Similitudes et différences avec les tableaux

Les points communs entre tableaux et ArrayLists sont nombreux :
— ils permettent de stocker une suite d'éléments de même type ;
— ils accèdent à leurs composantes via leur rang (indice) dans la collection ;
— les indices commencent à zéro et la même exception (`IndexOutOfBoundsException`) est levée lors d'un accès sur un indice invalide ;
— leurs types doivent spécifier le type de leurs composantes (tableau de String, ArrayList de String).

Les différences fondamentales sont :
— un tableau est de taille fixe, alors qu'un ArrayList change de taille selon qu'on y ajoute ou retire des éléments (redimensionement automatique, sans intervention de l'utilisateur) ;
— les opérations sur un tableau et sur un ArrayList sont très différentes ;
— les opérations disponibles sont bien plus nombreuses du côté des ArrayLists, sous formes de méthodes de la classe.

En programmation objet, à choisir entre tableau et ArrayList, on privilégie souvent ce dernier, pour la souplesse et la facilité d'utilisation, mais aussi parce que les ArrayLists sont des objets !

2.2 Import et création de base

Les ArrayLists se trouvent dans une bibliothèque qu'on doit importer dans notre programme. On doit déclarer en début de fichier :

```
import java.util.ArrayList;

public class ...
```

Un ArrayList est un objet, ce qui signifie qu'avant toute utilisation il doit être créé en mémoire via `new` suivi d'un constructeur. De plus, on doit utiliser la syntaxe objet pour invoquer les méthodes pertinentes sur la liste.

Cette syntaxe fait suivre un objet (une liste dans notre cas) d'un appel de méthode, séparés par un point : `objet.methode(...)`. Si `a` est un ArrayList, la syntaxe `a.size()` permet d'invoquer la méthode `size` sur la liste `a` pour obtenir sa taille.

Voici un premier exemple de création et d'utilisation :

```
// création d'une liste vide de chaînes de caractères
ArrayList<String> commandeReferences = new ArrayList<String>();
// afficher sa taille
System.out.print(commandeReferences.size()); // affiche 0
```

3 Type et création d'ArrayList

3.1 Déclaration avec un type spécifique

Un ArrayList doit contenir des éléments d'un même type. Contrairement aux tableaux qui peuvent contenir des types primitifs (comme `int` ou `double`), les ArrayLists ne peuvent contenir que des objets. Pour l'instant, nous nous concentrerons sur des types objets simples comme `String`.

La déclaration d'un ArrayList suit cette syntaxe :

```
ArrayList<String> commandeReferences;
ArrayList<String> nomsEtudiants;
ArrayList<String> listeVilles;
```

Le type entre chevrons (`< >`) indique le type des éléments que contiendra l'ArrayList. Ici, tous nos exemples contiennent des chaînes de caractères (`String`).

3.2 Création en mémoire avec new

Comme tout objet en Java, un ArrayList doit être créé en mémoire avant d'être utilisé. Cette création se fait avec le mot-clé `new` suivi d'un appel au constructeur :

```
// Déclaration et création en une seule instruction
ArrayList<String> commandeReferences = new ArrayList<String>();

// Ou séparément
ArrayList<String> nomsEtudiants;
nomsEtudiants = new ArrayList<String>();
```

Au moment de sa création, un ArrayList est toujours vide (il ne contient aucun élément). Sa taille est donc de 0.

3.3 Syntaxe d'appel des méthodes

Un ArrayList étant un objet, on utilise la syntaxe objet pour appeler ses méthodes. Cette syntaxe consiste à faire suivre le nom de l'objet d'un point, puis du nom de la méthode avec ses éventuels paramètres :

```
nomObjet.nomMethode(paramètres)
```

Par exemple :

```
ArrayList<String> liste = new ArrayList<String>();
int taille = liste.size(); // Appel de la méthode size()
liste.add("premier"); // Appel de la méthode add()
```

4 Les méthodes essentielles

Nous allons maintenant découvrir les méthodes les plus importantes d'un ArrayList, en commençant par celles qui permettent les opérations de base.

4.1 Méthodes d'information

int size() : cette méthode retourne le nombre d'éléments contenus dans l'ArrayList.

```
ArrayList<String> liste = new ArrayList<String>();
System.out.println(liste.size()); // affiche 0
```

boolean isEmpty() : cette méthode retourne `true` s'il n'y a aucun élément dans la liste, `false` sinon. C'est équivalent à tester si `size() == 0`.

```
ArrayList<String> liste = new ArrayList<String>();
if (liste.isEmpty()) {
   System.out.println("La liste est vide");
}
```

4.2 Ajout d'éléments

boolean add(String elem) : cette méthode ajoute l'élément `elem` à la fin de la liste. La taille de l'ArrayList augmente automatiquement de 1. La méthode retourne toujours `true`.

```
ArrayList<String> fruits = new ArrayList<String>();
fruits.add("pomme");
fruits.add("banane");
fruits.add("orange");
System.out.println(fruits.size()); // affiche 3
```

4.3 Accès aux éléments

String get(int i) : cette méthode retourne l'élément se trouvant à l'indice `i` de la liste. Comme pour les tableaux, les indices commencent à 0 et vont jusqu'à `size()-1`. Si l'indice n'est pas valide, une exception `IndexOutOfBoundsException` est levée.

```
ArrayList<String> fruits = new ArrayList<String>();
fruits.add("pomme");
fruits.add("banane");
fruits.add("orange");

System.out.println(fruits.get(0)); // affiche "pomme"
System.out.println(fruits.get(1)); // affiche "banane"
System.out.println(fruits.get(2)); // affiche "orange"
// fruits.get(3); // lèverait une exception
```

4.4 Premier exemple pratique

Voici un exemple complet qui illustre l'utilisation de ces méthodes essentielles :

```
public class ExempleArrayList {
   public static void main(String[] args) {
      // Création d'une liste de prénoms
      ArrayList<String> prenoms = new ArrayList<String>();

      // Ajout de quelques prénoms
      prenoms.add("Alice");
      prenoms.add("Bob");
      prenoms.add("Charlie");

      // Affichage de la taille
      System.out.println("Nombre de prénoms : " + prenoms.size());

      // Parcours et affichage de tous les prénoms
      for (int i = 0; i < prenoms.size(); i++) {
         System.out.println("Prénom " + i + " : " + prenoms.get(i));
      }

      // Test si la liste est vide
      if (!prenoms.isEmpty()) {
         System.out.println("Premier prénom : " + prenoms.get(0));
      }
   }
}
```

Ce programme produira la sortie suivante :

```
Nombre de prénoms : 3
Prénom 0 : Alice
Prénom 1 : Bob
Prénom 2 : Charlie
Premier prénom : Alice
```

5 Méthodes avancées de manipulation

Au-delà des opérations de base, ArrayList offre des méthodes plus sophistiquées pour manipuler les éléments : insertion à une position précise, suppression, remplacement et recherche.

5.1 Insertion à une position donnée

void add(int i, String elem) : cette méthode insère l'élément `elem` à l'indice `i` de la liste. Les éléments à partir de l'indice `i` voient leur indice augmenter de 1. L'indice `i` doit être valide (entre 0 et `size()` inclus). Si `i` égale `size()`, l'élément est ajouté en fin de liste.

```
ArrayList<String> couleurs = new ArrayList<String>();
couleurs.add("rouge");
couleurs.add("bleu");
couleurs.add("vert");
// État : ["rouge", "bleu", "vert"]

couleurs.add(1, "jaune"); // Insertion àl'indice 1
// État : ["rouge", "jaune", "bleu", "vert"]
System.out.println(couleurs.size()); // affiche 4
```

5.2 Suppression d'éléments

void remove(int i) : cette méthode retire l'élément d'indice `i`. Les éléments se trouvant à droite de `i` voient leur indice diminuer de 1. Si `i` n'est pas un indice valide, une exception `IndexOutOfBoundsException` est levée.

```
ArrayList<String> animaux = new ArrayList<String>();
animaux.add("chat");
animaux.add("chien");
animaux.add("oiseau");
// État : ["chat", "chien", "oiseau"]

animaux.remove(1); // Supprime "chien"
// État : ["chat", "oiseau"]
System.out.println(animaux.size()); // affiche 2
```

5.3 Remplacement d'éléments

String set(int i, String elem) : cette méthode remplace l'élément d'indice `i` par `elem`. Elle retourne l'ancien élément qui était à cette position. Si `i` n'est pas un indice valide, une exception `IndexOutOfBoundsException` est levée.

```
ArrayList<String> jours = new ArrayList<String>();
jours.add("lundi");
jours.add("mardi");
```

```
jours.add("mercredi");

String ancienJour = jours.set(1, "MARDI");
System.out.println("Ancien jour : " + ancienJour); // affiche "mardi"
// État : ["lundi", "MARDI", "mercredi"]
```

5.4 Recherche d'éléments

boolean contains(String o) : cette méthode retourne `true` si l'élément `o` est présent dans la liste, `false` sinon.

int indexOf(String o) : cette méthode retourne l'indice de la première occurrence de l'élément `o` dans la liste. Si l'élément n'est pas trouvé, elle retourne -1.

```
ArrayList<String> fruits = new ArrayList<String>();
fruits.add("pomme");
fruits.add("banane");
fruits.add("orange");
fruits.add("pomme"); // Doublon

// Test de présence
if (fruits.contains("banane")) {
   System.out.println("La banane est dans la liste");
}

// Recherche d'indice
int indice = fruits.indexOf("pomme");
System.out.println("Première pomme àl'indice : " + indice);
                 // affiche 0

int indiceIntrouvable = fruits.indexOf("kiwi");
System.out.println("Kiwi trouvé àl'indice : " + indiceIntrouvable);
                // affiche -1
```

5.5 Exemple avec modifications

Voici un exemple qui illustre l'utilisation de ces méthodes avancées :

```
public class GestionListe {
   public static void main(String[] args) {
      ArrayList<String> taches = new ArrayList<String>();

      // Ajout de tâches
      taches.add("Faire les courses");
      taches.add("Nettoyer la maison");
      taches.add("Réviser le cours");

      System.out.println("Tâches initiales :");
      afficherListe(taches);

      // Insertion d'une tâche urgente en première position
      taches.add(0, "Appeler le médecin");

      // Modification d'une tâche
      taches.set(2, "Faire les courses au supermarché");
```

```
        // Suppression d'une tâche terminée
        if (taches.contains("Nettoyer la maison")) {
            int indice = taches.indexOf("Nettoyer la maison");
            taches.remove(indice);
        }

        System.out.println("\nTâches après modifications :");
        afficherListe(taches);
    }

    public static void afficherListe(ArrayList<String> liste) {
        for (int i = 0; i < liste.size(); i++) {
            System.out.println((i+1) + ". " + liste.get(i));
        }
    }
}
```

6 Itération sur ArrayList

Les boucles de parcours d'un ArrayList suivent souvent le même schéma que celles de parcours de tableaux. Il s'agit généralement d'une boucle `for` qui itère sur tout ou sur une partie des indices valides, obtient l'élément à chaque indice avec `get(i)`, et réalise des traitements sur cet élément.

6.1 Parcours complet

La forme la plus courante de parcours d'un ArrayList est le parcours complet de tous ses éléments :

```
ArrayList<String> villes = new ArrayList<String>();
villes.add("Paris");
villes.add("Lyon");
villes.add("Marseille");

// Parcours complet
for (int i = 0; i < villes.size(); i++) {
    System.out.println("Ville " + i + " : " + villes.get(i));
}
```

6.2 Traitement des éléments

Voici un exemple plus complexe qui calcule la longueur totale de toutes les chaînes contenues dans un ArrayList :

```
public static int calculerLongueurTotale(ArrayList<String> mots) {
    int longueurTotale = 0;

    for (int i = 0; i < mots.size(); i++) {
        String motCourant = mots.get(i);
        longueurTotale = longueurTotale + motCourant.length();
    }

    return longueurTotale;
}
```

6.3 Parcours avec condition

On peut aussi parcourir un ArrayList en appliquant des conditions :

```
public static void afficherMotsLongs(ArrayList<String> mots,
                              int longueurMin) {
   System.out.println("Mots de plus de " + longueurMin +
                  " caractères :");

   for (int i = 0; i < mots.size(); i++) {
      String motCourant = mots.get(i);
      if (motCourant.length() > longueurMin) {
         System.out.println("- " + motCourant);
      }
   }
}
```

6.4 Exemple complet : statistiques sur une liste de mots

```
public class StatistiquesMots {
   public static void main(String[] args) {
      ArrayList<String> mots = new ArrayList<String>();
      mots.add("bonjour");
      mots.add("au");
      mots.add("revoir");
      mots.add("et");
      mots.add("à");
      mots.add("bientôt");

      // Calculs statistiques
      int nombreMots = mots.size();
      int longueurTotale = calculerLongueurTotale(mots);
      double longueurMoyenne = (double) longueurTotale / nombreMots;

      System.out.println("Statistiques :");
      System.out.println("Nombre de mots : " + nombreMots);
      System.out.println("Longueur totale : " + longueurTotale);
      System.out.println("Longueur moyenne : " + longueurMoyenne);

      // Affichage des mots longs
      afficherMotsLongs(mots, 4);
   }

   public static int calculerLongueurTotale(ArrayList<String> mots) {
      int longueurTotale = 0;
      for (int i = 0; i < mots.size(); i++) {
         longueurTotale += mots.get(i).length();
      }
      return longueurTotale;
   }

   public static void afficherMotsLongs(ArrayList<String> mots,
                                 int longueurMin) {
      System.out.println("Mots de plus de " + longueurMin +
```

```
                    " caractères :");
        for (int i = 0; i < mots.size(); i++) {
            String motCourant = mots.get(i);
            if (motCourant.length() > longueurMin) {
                System.out.println("- " + motCourant);
            }
        }
    }
}
```

Ce programme produira la sortie suivante :

```
Statistiques :
Nombre de mots : 6
Longueur totale : 28
Longueur moyenne : 4.666666666666667
Mots de plus de 4 caractères :
- bonjour
- revoir
- bientôt
```

6.5 Boucle for-each

Java propose une syntaxe alternative pour parcourir les collections : la boucle `for-each` (aussi appelée "enhanced for loop"). Cette syntaxe est plus simple et plus lisible quand on veut parcourir tous les éléments d'un ArrayList sans avoir besoin de connaître leur indice. La syntaxe générale est :

```
for (String element : arraylist) {
// traitement de element
}
```

Cette boucle se lit : "pour chaque élément de type String dans l'arraylist". À chaque itération, la variable `element` contient l'élément courant. Voici une comparaison entre les deux approches :

```
ArrayList<String> fruits = new ArrayList<String>();
fruits.add("pomme");
fruits.add("banane");
fruits.add("orange");
// Avec une boucle for classique
System.out.println("Avec boucle for classique :");
for (int i = 0; i < fruits.size(); i++) {
   String fruit = fruits.get(i);
   System.out.println(fruit);
}
// Avec une boucle for-each
System.out.println("\nAvec boucle for-each :");
for (String fruit : fruits) {
   System.out.println(fruit);
}
```

La boucle `for-each` est particulièrement utile pour des traitements simples :

```
public static int calculerLongueurTotaleForEach(ArrayList<String> mots) {
  int longueurTotale = 0;
  for (String mot : mots) {
    longueurTotale += mot.length();
```

```
    }

    return longueurTotale;
}
```

Limitations de la boucle for-each :

- On ne peut pas modifier la structure de l'ArrayList (ajouter ou supprimer des éléments) pendant le parcours
- On n'a pas accès à l'indice de l'élément courant
- On ne peut parcourir que dans l'ordre (du début vers la fin)

La boucle `for-each` est donc idéale pour la lecture et les calculs, tandis que la boucle `for` classique reste nécessaire quand on a besoin de l'indice ou quand on veut modifier la liste. Ce programme produira la sortie suivante :

```
Statistiques :
Nombre de mots : 6
Longueur totale : 28
Longueur moyenne : 4.666666666666667
Mots de plus de 4 caractères :

bonjour
revoir
bientôt
```

7 Exemple complet : Catalogue d'articles

Dans cette section, nous présentons un exemple complet utilisant ArrayList pour modéliser un catalogue d'articles. Cet exemple illustre une approche orientée objet en combinant une classe métier (Article) avec une collection ArrayList.

7.1 La classe Article

Un article est caractérisé par sa référence, sa dénomination et son prix. Nous définissons un constructeur pour initialiser les attributs d'un article à sa création.

```
public class Article {
  private String reference;
  private String denomination;
  private double prix;
  public Article(String ref, String den, double p) {
    this.reference = ref;
    this.denomination = den;
    this.prix = p;
  }

  public double getPrix() {
    return prix;
  }

  public void setPrix(double prix) {
    this.prix = prix;
  }

  public String getReference() {
    return reference;
```

```
  }

  public String getDenomination() {
    return denomination;
  }

  public void setDenomination(String dn) {
    denomination = dn;
  }

  @Override
  public String toString() {
    return "Article[reference=" + reference + ", denomination=" +
          denomination + ", prix=" + prix + "]";
  }
}
```

Notez qu'il n'y a pas de méthode pour modifier la référence d'un article, mais seulement pour modifier son prix ou sa dénomination. La méthode `toString` est redéfinie pour permettre un affichage lisible des articles.

7.2 Le catalogue d'articles

Nous voulons modéliser un catalogue d'articles avec des opérations pour ajouter et retirer un article du catalogue, pour afficher le catalogue, et pour calculer le prix à payer pour un panier d'achats donné par une liste de références. Les articles du catalogue sont identifiés par leur référence, et nous devons éviter qu'il y ait plusieurs articles avec la même référence.

7.2.1 Structure principale et initialisation

```
public static void main(String[] args) {
  // Création d'un article
  Article riz3 = new Article("Riz3", "Riz Toto premium", 5.0);
  // Déclaration, création et initialisation du catalogue
  ArrayList<Article> cat = initCatalogue();

  // Affichage
  afficheCatalogue(cat);

  // Retirer une référence
  retireReference("Sauce2", cat);
  ajoutCatalogue(riz3, cat);
  afficheCatalogue(cat);

  // Calcul et affichage du prix d'un panier
  ArrayList<String> panier = new ArrayList<String>();
  panier.add("Riz1");
  panier.add("PT1");
  double apayer = prixPanier(panier, cat);
  System.out.println("Prix àpayer = " + apayer);
}
/**

Retourne un catalogue contenant 4 articles.
```

```
*/
public static ArrayList<Article> initCatalogue() {
  ArrayList<Article> l = new ArrayList<Article>();
  l.add(new Article("Riz1", "Riz Toto", 3.5));
  l.add(new Article("Sauce1", "Sauce Titi", 5.6));
  l.add(new Article("PT1", "Pate tartiner noisette", 10.5));
  l.add(new Article("Sauce2", "Sauce Tutu", 10.0));
  return l;
}

/**

Affiche les articles d'un catalogue.
*/
public static void afficheCatalogue(ArrayList<Article> cat) {
  System.out.println(cat.toString());
}
```

7.2.2 Recherche d'une référence

Les articles étant identifiés par leur référence, nous commençons par écrire une méthode qui cherche une référence dans un catalogue et qui retourne l'indice de l'article correspondant si elle se trouve dans le catalogue, et -1 sinon.

```
/**

Retourne l'indice d'une référence du catalogue si elle existe, -1
sinon.
*/
public static int indiceReference(String ref, ArrayList<Article> cat) {
  for (int i = 0; i < cat.size(); i++) {
    if (cat.get(i).getReference().equals(ref)) {
      return i;
    }
  }
  return -1;
}
```

7.2.3 Ajout et retrait

```
/**

Ajoute a au catalogue si un article de même référence n'existe pas
au catalogue, lève une exception sinon.
*/
public static void ajoutCatalogue(Article a, ArrayList<Article> cat) {
  int idx = indiceReference(a.getReference(), cat);
  if (idx == -1) { // ajout si référence non trouvée
    cat.add(a);
  } else {
    throw new IllegalArgumentException("Article déjà présent");
  }
}
```

```
/**

Retire une référence du catalogue si elle existe.
*/
public static void retireReference(String ref, ArrayList<Article> cat) {
  int idx = indiceReference(ref, cat);
  if (idx != -1) { // retirée si trouvée
  cat.remove(idx);
}
```

7.2.4 Prix d'un panier d'achats

```
/**

Retourne le prix total d'un panier d'articles donné par une liste de
références. Ignore les références inexistantes au catalogue.
*/
public static double prixPanier(ArrayList<String> pn,
                        ArrayList<Article> cat) {
  double res = 0;
  for (int i = 0; i < pn.size(); i++) {
    int idx = indiceReference(pn.get(i), cat);
    if (idx >= 0) {
      res = res + cat.get(idx).getPrix();
    }
  }
  return res;
}
```

8 Notions avancées

8.1 Généricité avec ArrayList<E>

Jusqu'à présent, nous avons utilisé des ArrayList de String, mais ArrayList peut contenir n'importe quel type d'objet. Dans la documentation Java, vous verrez souvent la notation `ArrayList<E>`, où E représente un type d'objet quelconque. Cette notation est générique dans le sens où :

1. on peut stocker des éléments de n'importe quel type objet E, qui doit être le même pour tous les éléments
2. les méthodes sur les ArrayList fonctionnent indépendamment du type de leurs éléments

Voici quelques exemples de déclarations avec différents types :

```
ArrayList<String> noms; // E = String
ArrayList<Article> catalogue; // E = Article
ArrayList<Integer> nombres; // E = Integer
```

Quand vous lisez la documentation et voyez une méthode comme `E get(int i)`, cela signifie que la méthode retourne un objet du type E déclaré pour votre ArrayList.

8.2 Types primitifs vs types objets

ArrayList ne peut contenir que des objets, pas des types primitifs comme `int`, `double`, `boolean` ou `char`. Java fournit des classes "wrapper" pour chaque type primitif :

Type primitif	Classe wrapper
int	Integer
double	Double
boolean	Boolean
char	Character

Depuis Java 5, il existe un mécanisme de conversion automatique (autoboxing/unboxing) entre les types primitifs et leurs classes wrapper :

```
ArrayList<Integer> nombres = new ArrayList<Integer>();
// Ces deux lignes sont équivalentes :
nombres.add(42); // conversion automatique int -> Integer
nombres.add(new Integer(42)); // création explicite d'Integer
// De même pour la lecture :
int valeur = nombres.get(0); // conversion automatique Integer -> int
```

8.3 Bonnes pratiques

- **Préférez ArrayList aux tableaux** quand la taille de votre collection peut varier
- **Utilisez des noms de variables explicites** : `listeEtudiants` plutôt que `liste`
- **Encapsulez vos ArrayList** dans des classes métier plutôt que de les exposer directement
- **Vérifiez les indices** avant d'accéder aux éléments si vous n'êtes pas sûr de leur validité
- **Utilisez for-each** pour les parcours simples, for classique quand vous avez besoin de l'indice
- **Documentez vos méthodes** qui manipulent des ArrayList, en précisant le comportement en cas d'erreur

9 Conclusion

La classe ArrayList constitue un outil fondamental en programmation Java pour gérer des collections de données de taille variable. Elle offre une alternative puissante et flexible aux tableaux traditionnels, particulièrement adaptée aux situations où le nombre d'éléments n'est pas connu à l'avance ou peut évoluer durant l'exécution du programme.

Les principaux avantages d'ArrayList par rapport aux tableaux sont sa capacité de redimensionnement automatique, ses nombreuses méthodes de manipulation intégrées, et sa syntaxe orientée objet qui rend le code plus lisible et maintenable. La richesse de son API permet de réaliser facilement des opérations courantes comme la recherche, l'insertion, la suppression, ou le parcours d'éléments.

L'exemple du catalogue d'articles illustre comment ArrayList s'intègre naturellement dans une approche orientée objet, en permettant de construire des applications robustes qui manipulent des collections d'objets métier. Cette approche est représentative de nombreux problèmes réels en développement logiciel.

La maîtrise d'ArrayList constitue une base solide pour aborder d'autres structures de données plus complexes du framework Java Collections, comme LinkedList, HashSet ou HashMap. Les concepts et techniques présentés dans ce chapitre - gestion d'indices, parcours avec boucles, manipulation d'objets dans des collections - sont transposables à l'ensemble de ces structures.

En pratique, ArrayList sera l'une des classes les plus fréquemment utilisées dans vos programmes Java. Sa simplicité d'utilisation ne doit pas masquer l'importance de bien comprendre

ses mécanismes internes et ses limites, notamment en termes de performance pour des collections très volumineuses ou des opérations d'insertion/suppression fréquentes en milieu de liste.

Ce chapitre a été rédigé avec l'assistance d'une intelligence artificielle, Claude 4 Sonnet de la société Anthropic