

# Projet Java : Listes dynamiques et triées

## Étude des piliers de la Programmation Orientée Objet

### Objectif pédagogique

Ce projet a pour but d'illustrer les **quatre piliers de la Programmation Orientée Objet (POO)** :

1. **Encapsulation**
2. **Héritage**
3. **Polymorphisme**
4. **Abstraction**

L'objectif est de concevoir une hiérarchie de classes représentant des listes dynamiques avec différents comportements, tout en respectant les principes de modularité, de réutilisation du code et de robustesse.

# 1 Partie 1 — Classe de base SimpleList

## Description

On souhaite implémenter une structure de données de type **liste dynamique** simplifiée, qui encapsule un tableau interne et offre des opérations de base.

## Spécifications

Créer une classe publique `SimpleList` ayant :

- `int capacity` : capacité maximale de la liste (privé).
- `int length` : nombre d'éléments actuellement stockés (privé).
- `int[] elements` : tableau contenant les éléments (privé).

## Constructeurs

- `SimpleList(int capacity)` : crée une liste vide avec la capacité donnée.

## Méthodes publiques

- `int getLength()` : retourne la longueur actuelle.
- `int getCapacity()` : retourne la capacité.
- `void append(int value)` : ajoute un élément à la fin si la capacité le permet.
- `void remove(int index)` : supprime l'élément à l'indice indiqué.
- `String toString()` : retourne une chaîne représentant le contenu de la liste.

## Comportement attendu

Les attributs doivent être **privés** et accessibles uniquement via des accesseurs (→ **encapsulation**). Les méthodes doivent vérifier la validité des opérations (lever une exception si nécessaire).

## Exemple d'utilisation

```
1 SimpleList list = new SimpleList(5);
2 list.append(10);
3 list.append(3);
4 list.append(7);
5 System.out.println(list); // [10, 3, 7]
6 list.remove(1);
7 System.out.println(list); // [10, 7]
```

## 2 Partie 2 — Classe dérivée SortedList

### Description

On souhaite maintenant créer une version spécialisée de la liste : une **liste toujours triée**. Cette nouvelle classe réutilisera la structure et les méthodes de `SimpleList`, mais modifiera le comportement d'ajout.

### Spécifications

Créer une classe `SortedList` qui **hérite de** `SimpleList`.

### Constructeur

- `SortedList(int capacity)` : crée une liste vide triée.

### Méthodes redéfinies

- `void append(int value)` : devient un **ajout intelligent** :
  - Si la valeur n'existe pas → insère l'élément à sa place pour conserver l'ordre croissant.
  - Si la valeur existe → **ne fait rien** (l'élément n'est pas ajouté).
- `String toString()` : reste **identique** à la méthode de la classe mère.

### Nouvelle méthode

- `void removeElement(int value)` : **nouvelle méthode** qui supprime un élément par sa valeur :
  - Si la valeur existe dans la liste → supprime l'élément.
  - Si la valeur n'existe pas → **ne fait rien**.

### Héritage

La méthode `remove(int index)` est **héritée sans modification**. L'attribut `elements` restant privé, la sous-classe doit y accéder via des **méthodes protégées** ou des accesseurs adaptés.

### Exemple d'utilisation

```
1 SortedList sList = new SortedList(10);
2 sList.append(4);
3 sList.append(2);
4 sList.append(7);
5 sList.append(4); // 4 existe déjà, donc aucun changement
6 System.out.println(sList); // [2, 4, 7]
7 sList.removeElement(4); // Supprime l'élément 4
8 System.out.println(sList); // [2, 7]
9 sList.removeElement(10); // 10 n'existe pas, donc aucun changement
10 System.out.println(sList); // [2, 7]
```

## 3 Partie 3 — Abstraction et polymorphisme

### Objectif

Introduire une **classe abstraite** pour généraliser les comportements communs des listes.

### Spécifications

Créer une classe abstraite `AbstractList` :

- Attributs protégés : `capacity`, `length`
- Méthodes abstraites :
  - `void append(int value)`
  - `void remove(int index)`
- Méthodes concrètes :
  - `int getLength()`
  - `int getCapacity()`

Faire en sorte que :

- `SimpleList` hérite de `AbstractList` et implémente ses méthodes.
- `SortedList` hérite de `SimpleList`.

### Test du polymorphisme

```
1 public static void printList(AbstractList list) {  
2     System.out.println(list.toString());  
3 }  
4  
5 printList(new SimpleList(5));  
6 printList(new SortedList(5));
```

## 4 Partie 4 — Extensions possibles (bonus)

- Créer une classe `UniqueList` (interdit les doublons, héritée de `SimpleList`).
- Créer une classe `ResizableList` (augmente automatiquement sa capacité quand elle est pleine).
- Surcharger des opérateurs classiques (`equals`, `compareTo`).
- Créer une interface `Removable` avec la méthode `remove(int index)` et la faire implémenter par `SimpleList`.
- Ajouter une interface `Searchable` avec des méthodes de recherche avancée.

## Compétences travaillées

- Encapsulation : protection des données internes.
- Héritage : réutilisation et spécialisation du code.
- Polymorphisme : appel dynamique de méthodes.
- Abstraction : conception de classes et méthodes génériques.
- Composition et modularité (bonus).