CMSC 132: OBJECT-ORIENTED PROGRAMMING II

List Interface and the Vector Class

Department of Computer Science
University of Maryland, College Park

Introduction to the List Interface

What is the List Interface?

- The List interface in Java is part of the **Java Collections Framework (JCF)**.
- Represents an **ordered collection** (also called a sequence) of elements.
- Allows **duplicates** and provides precise control over where each element is inserted.
- Serves as the **Abstract Data Type (ADT)** representation for a list in Java.

Introduction to the List Interface

- **ArrayList, Vector, and Stack** are all implementations of the List interface.
- **ArrayList**: A widely used implementation backed by a dynamic array. (We saw this in lecture)

  https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ArrayList.html
- **Vector**: Similar to ArrayList, but grows dynamically by a larger factor and is synchronized. (We will learn about this today)

  https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Vector.html
- **Stack**: Extends Vector and represents a stack (LIFO) data structure. (We saw this in lecture)

  https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Stack.html

Characteristics of the List Interface

- **Order**: Elements in a List are maintained in the order they were inserted.
- **Indexing**: Elements can be accessed using an index (similar to arrays).
- **Common Operations**:add(), get(), remove(), size(), clear(), contains()

Vector Class Overview

What is a Vector?

- A **resizable array** implementation of the List interface.
- Grows dynamically as elements are added (typically by doubling its size).
- **Synchronized**: Threads can safely modify a Vector concurrently.

Key Characteristics of Vector:

- **Size Growth**: When the vector is full, it grows by 100% (doubling in size) by default.
- **Legacy Class**: Originally part of the Java 1.0 version, now considered outdated for many use cases (replaced by ArrayList).

Comparing Vector vs ArrayList

Feature	Vector	ArrayList
Synchronized	Yes (thread-safe)	No (not thread-safe)
Growth Policy	Doubles the size when full	Increases size by 50% by default
Performance	Slower due to synchronization overhead	Faster (no synchronization)
Default Size	10 elements	10 elements
Resizing Behavior	Growth factor is 100%	Growth factor is 50%
Legacy Status	Legacy (older version of List)	Preferred modern implementation

Methods Unique to Vector, ArrayList, and Stack

- **Methods in Vector** (not in List interface):
  - addElement(E obj): Adds an element to the vector (replaces add()).
  - elementAt(int index): Retrieves an element at the specified index.
  - removeElement(Object obj): Removes the first occurrence of the specified element.
  - capacity(): Returns the current capacity of the vector.
  - trimToSize(): Resizes the vector to the current size.
- **Methods in ArrayList** (not in List interface):
  - ensureCapacity(int minCapacity): Ensures that the list can hold at least the specified number of elements.
  - trimToSize(): Reduces the size of the internal array to match the number of elements.
- **Methods in Stack** (not in List interface):
  - push(E item): Pushes an item onto the stack.
  - pop(): Removes and returns the top element from the stack.
  - peek(): Returns the top element without removing it.

Use Cases for Each Implementation

- **ArrayList**:
  - Most commonly used for general-purpose storage when synchronization is not required.
  - Ideal for random access and when list sizes fluctuate but are generally not large.
- **Vector**:
  - Used in legacy code or situations where synchronization is required and performance is not a major concern.
  - Less commonly used today in favor of ArrayList and CopyOnWriteArrayList (for thread-safety).
- **Stack**:
  - Used when the application needs to follow **Last In First Out (LIFO)** behavior.
  - Ideal for situations like **undo/redo** functionality or **parsing expressions**.

When to Use Each Class

- **Use ArrayList**:
  - When you need a general-purpose list and thread safety is not a concern.
  - When you need fast access to elements and the list size may change frequently.
- **Use Vector**:
  - When you require thread-safe behavior (though ArrayList with external synchronization is a better choice in modern applications).
  - For legacy applications that still rely on Vector.
- **Use Stack**:
  - When your use case requires stack behavior (LIFO order), such as function calls or undo/redo operations.

See list package in LabWeek3 Project