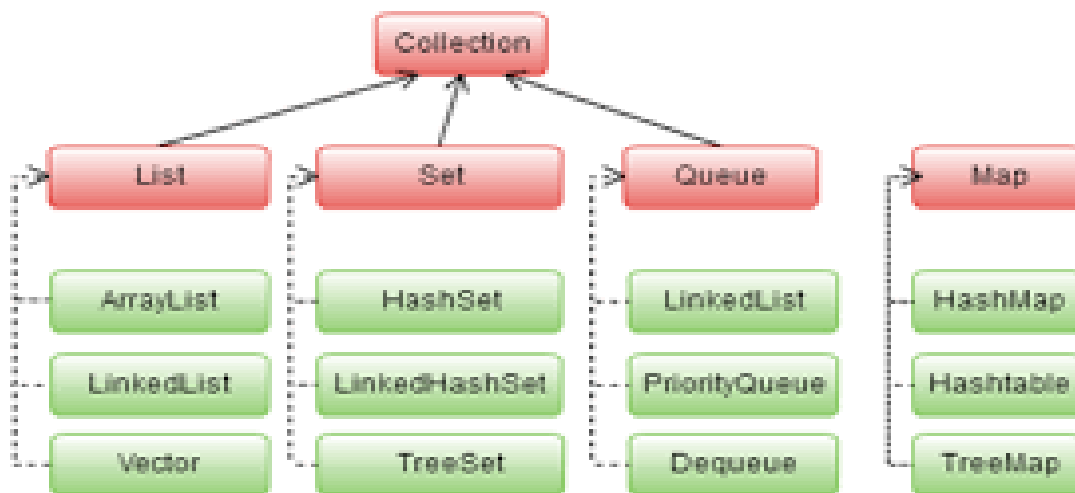


Introduction to Collection Framework

- The **Java Collection Framework (JCF)** is a set of **classes and interfaces** that provide ready-made data structures and algorithms.
- It is present in `java.util` package.
- It provides interfaces like `List`, `Set`, `Queue`, `Map` and their implementations.
- Benefits:
 - Reduces programming effort.
 - Provides reusable, well-tested data structures.
 - Supports algorithms like sorting, searching, etc.



1. List Interface

- **Definition:**

The **List Interface** is part of the *Java Collection Framework*. It represents an **ordered collection** (also known as a **sequence**) that allows storing **duplicate elements**.
- **Key Characteristics:**
 - **Ordered Collection** → Elements are stored in the order they are inserted.
 - **Allows Duplicates** → Unlike `Set`, a `List` can have multiple elements with the same value.

- **Index-based Access** → Elements can be accessed, inserted, or removed using their position (like arrays, but more flexible).
- **Dynamic Size** → Unlike arrays (fixed size), lists can grow or shrink dynamically.
- **Important Implementations:**
 - **ArrayList** → Fast for random access, not synchronized.
 - **LinkedList** → Good for frequent insertions/deletions.
 - **Vector** → Synchronized version of ArrayList (legacy).
 - **Stack** → A subclass of Vector, follows **LIFO (Last In, First Out)**.

(I) ArrayList (Resizable Array)

- ArrayList is a **resizable array** implementation of the List interface.
- Allows **random access** (fast get / set).
- Slower for **insertions/deletions** in the middle (since shifting occurs).
- Allows **duplicate elements** and maintains **insertion order**.

Important Methods

- `add(E e)` → Appends element at end.
- `add(int index, E e)` → Insert at given index.
- `get(int index)` → Retrieve element.
- `set(int index, E e)` → Replace element.
- `remove(int index) / remove(Object o)` → Remove element.
- `contains(Object o)` → Check if element exists.
- `size()` → Number of elements.
- `indexOf(Object o) / lastIndexOf(Object o)` → Position of element.
- `clear()` → Remove all elements.

Example

```
import java.util.ArrayList;

public class ArrayListDemo {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");
    }
}
```

```

        list.add("C");
        System.out.println("Original: " + list);

        list.set(1, "Z");    // update
        list.remove("A");    // remove by value
        System.out.println("Updated: " + list);
    }
}

```

(II) LinkedList (Doubly Linked List)

- LinkedList implements both **List** and **Deque** interfaces.
- Uses a **doubly linked list** internally.
- Best for **frequent insertions and deletions**.
- Slower for **random access** compared to ArrayList.

Important Methods

- `addFirst(E e) / addLast(E e)` → Insert at start/end.
- `getFirst() / getLast()` → Retrieve first/last element.
- `removeFirst() / removeLast()` → Remove first/last.
- `peek() / peekFirst() / peekLast()` → View element (no removal).
- `poll() / pollFirst() / pollLast()` → Retrieve + remove element.
- `offer(E e)` → Insert like in a queue.

Example

```

import java.util.LinkedList;

public class LinkedListDemo {
    public static void main(String[] args) {
        LinkedList<Integer> ll = new LinkedList<>();
        ll.add(10);
        ll.add(20);
        ll.addFirst(5);
        ll.addLast(30);
        System.out.println("LinkedList: " + ll);
    }
}

```

```

        ll.removeFirst();
        System.out.println("After removal: " + ll);
    }
}

```

(III) Vector (Legacy, Thread-Safe)

- Vector is a **legacy class** (introduced in JDK 1.0).
- Similar to ArrayList, but **synchronized** (thread-safe).
- Slower in single-threaded applications compared to ArrayList.

Important Methods

- `add(E e) / addElement(E e)` → Add element.
- `elementAt(int index)` → Like `get()`.
- `firstElement() / lastElement()` → Retrieve ends.
- `removeElement(Object o)` → Remove by value.
- `removeAllElements()` → Clears all elements.
- `capacity()` → Current storage capacity.

Example

```

import java.util.Vector;

public class VectorDemo {
    public static void main(String[] args) {
        Vector<String> v = new Vector<>();
        v.add("Java");
        v.add("Python");
        v.add("C++");

        System.out.println("Vector: " + v);
        System.out.println("First: " + v.firstElement());
        System.out.println("Last: " + v.lastElement());
        System.out.println("Capacity: " + v.capacity());
    }
}

```

```
}
```

(IV) Stack (LIFO – Legacy)

- Stack is a **subclass of Vector**.
- Implements **LIFO (Last In First Out)** data structure.
- Useful in **undo/redo, expression evaluation, recursion calls**.

Important Methods

- `push(E e)` → Insert element at top.
- `pop()` → Remove + return top element.
- `peek()` → Return top element without removing.
- `search(Object o)` → 1-based position from top.
- `empty()` → Check if stack is empty.

Example

```
import java.util.Stack;

public class StackDemo {
    public static void main(String[] args) {
        Stack<Integer> st = new Stack<>();
        st.push(10);
        st.push(20);
        st.push(30);

        System.out.println("Stack: " + st);
        System.out.println("Peek: " + st.peek());
        System.out.println("Pop: " + st.pop());
        System.out.println("After Pop: " + st);
    }
}
```