Java Collections Framework - Step by Step Guide 🚀

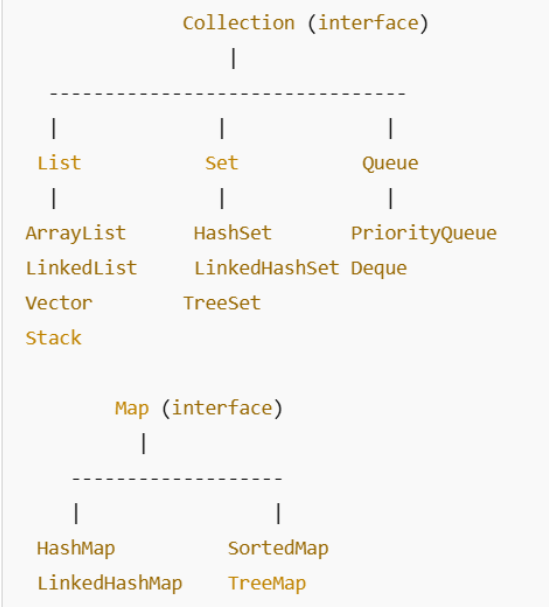
◆ What is Java Collections Framework?

The **Java Collections Framework (JCF)** is a set of classes and interfaces that **store and manipulate groups of objects** efficiently. It provides **ready-made data structures** such as **Lists, Sets, Queues, and Maps**.

◆ Collection Hierarchy in Java

🛠 Main Interfaces of Java Collections:

1) **List** – Ordered collection (allows duplicates)
2) **Set** – Unique elements (no duplicates)
3) **Queue** – FIFO (First-In-First-Out) processing
4) **Map** – Key-Value pairs (not a part of `Collection` interface)



◆ What is a List in Java?

A **List** in Java is an **ordered** collection that allows **duplicate elements** and provides **indexed access** to its elements.

✅ **Maintains insertion order**
✅ **Allows duplicate elements**
✅ **Can access elements by index (`list.get(index)`)**
✅ **Supports dynamic resizing** (Unlike arrays, which have a fixed size)

🔷 List Interface Hierarchy

mathematica Copy Edit

```
          List (interface)
              |
  ----------------------------
  |            |             |
ArrayList    LinkedList    Vector (Thread-Safe)
                              |
                           Stack (LIFO)
```

🚀 Example: `ArrayList`

java

Copy Edit

```java
import java.util.*;

public class ListExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");
        names.add("Alice"); // Duplicates allowed

        System.out.println("ArrayList: " + names);
        System.out.println("First Element: " + names.get(0)); // Access via index
    }
}
```

🔷 Output:

yaml

Copy Edit

```yaml
ArrayList: [Alice, Bob, Charlie, Alice]
First Element: Alice
```

2 LinkedList Class Methods & Operations

Operation	Method	Time Complexity
Add at End	`add(E e)`	`O(1)`
Add at Index	`add(int index, E e)`	`O(n)`
Remove First	`removeFirst()`	`O(1)`
Remove Last	`removeLast()`	`O(1)`
Remove by Index	`remove(int index)`	`O(n)`
Get First Element	`getFirst()`	`O(1)`
Get Last Element	`getLast()`	`O(1)`
Get by Index	`get(int index)`	`O(n)`
Check if Empty	`isEmpty()`	`O(1)`
Find Element	`contains(Object o)`	`O(n)`

📌 Example: Using `LinkedList` as a List

java Copy Edit

```java
import java.util.LinkedList;

public class LinkedListAsList {
    public static void main(String[] args) {
        LinkedList<String> names = new LinkedList<>();

        // Add elements at the end
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        // Add at specific index
        names.add(1, "David");

        // Display list
        System.out.println("LinkedList: " + names);

        // Get first & last elements
        System.out.println("First: " + names.getFirst());
        System.out.println("Last: " + names.getLast());

        // Remove elements
        names.remove("Bob");
        names.removeFirst();
        names.removeLast();

        System.out.println("Updated LinkedList: " + names);
    }
}
```

📌 Output:

yaml Copy Edit

```yaml
LinkedList: [Alice, David, Bob, Charlie]
First: Alice
Last: Charlie
Updated LinkedList: [David]
```

📌 Example: Stack Operations in Java

java Copy Edit

```java
import java.util.Stack;

public class StackExample {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();

        // Pushing elements onto stack
        stack.push(10);
        stack.push(20);
        stack.push(30);

        System.out.println("Stack: " + stack);

        // Peek operation (top element)
        System.out.println("Top element: " + stack.peek()); // 30

        // Pop operation (remove top element)
        System.out.println("Popped: " + stack.pop()); // 30

        // Checking if stack is empty
        System.out.println("Is Stack Empty? " + stack.isEmpty()); // false

        // Searching for an element (1-based index)
        System.out.println("Position of 10: " + stack.search(10)); // 2 (position from top)

        System.out.println("Final Stack: " + stack);
    }
}
```

📌 **Output:**

yaml Copy Edit

```yaml
Stack: [10, 20, 30]
Top element: 30
Popped: 30
Is Stack Empty? false
Position of 10: 2
Final Stack: [10, 20]
```

Set Interface in Java (Ordered, Unordered, Duplicates Not Allowed) 🚀

The Set interface in Java represents a **collection that does not allow duplicate elements**. It is part of the Java Collections Framework (java.util.Set<E>).

1. Key Features of Set Interface

✅ **No Duplicates Allowed** (Ensures unique elements).
✅ **Allows at most one null value** (Except TreeSet, which does not allow null).
✅ **Does Not Guarantee Order** (Except LinkedHashSet and TreeSet).

2 Implementations of `Set<E>`

Implementation	Ordering	Duplicate Allowed?	Performance (Avg.)
`HashSet<E>`	❌ Unordered	❌ No	✅ **O(1) (Fastest for add, remove, contains)**
`LinkedHashSet<E>`	✅ Insertion Order	❌ No	✅ `O(1)`
`TreeSet<E>`	✅ Sorted Order (Ascending)	❌ No	❌ `O(log n)` (Slower, uses Red-Black Tree)

📌 Example: Using `HashSet`

java

Copy Edit

```java
import java.util.HashSet;

public class HashSetExample {
    public static void main(String[] args) {
        HashSet<Integer> set = new HashSet<>();

        // Adding elements
        set.add(10);
        set.add(20);
        set.add(30);
        set.add(10); // Duplicate, ignored

        System.out.println("HashSet: " + set);

        // Check if an element exists
        System.out.println("Contains 20? " + set.contains(20));

        // Remove an element
        set.remove(10);

        System.out.println("Updated HashSet: " + set);
    }
}
```

📌 Output:

sql

Copy Edit

```sql
HashSet: [20, 10, 30]  (Order may vary)
Contains 20? true
Updated HashSet: [20, 30]
```

✅ **Best For:** Fast insertions, deletions, and searches (`O(1)`).

❌ **Not Ordered** (Elements are stored randomly).

📌 Example: Using `LinkedHashSet`

java Copy Edit

```java
import java.util.LinkedHashSet;

public class LinkedHashSetExample {
    public static void main(String[] args) {
        LinkedHashSet<String> set = new LinkedHashSet<>();

        // Adding elements
        set.add("Apple");
        set.add("Banana");
        set.add("Cherry");

        System.out.println("LinkedHashSet: " + set);

        // Removing an element
        set.remove("Banana");

        System.out.println("Updated LinkedHashSet: " + set);
    }
}
```

📌 Output:

yaml Copy Edit

```yaml
LinkedHashSet: [Apple, Banana, Cherry]
Updated LinkedHashSet: [Apple, Cherry]
```

✅ **Best For:** Maintaining insertion order.

❌ **Slightly slower (`O(1)`) than `HashSet` due to linked list overhead.**

📌 Example: Using `TreeSet`

java

Copy Edit

```java
import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<Integer> set = new TreeSet<>();

        // Adding elements
        set.add(30);
        set.add(10);
        set.add(20);

        System.out.println("TreeSet: " + set); // Sorted order

        // First & Last Elements
        System.out.println("First: " + set.first());
        System.out.println("Last: " + set.last());
    }
}
```

📌 Output:

makefile

Copy Edit

```makefile
TreeSet: [10, 20, 30]
First: 10
Last: 30
```

✅ **Best For:** Sorted elements in natural order (`O(log n)`).

❌ **Slower than** `HashSet` (`O(log n)`) **due to tree operations.**

6 Performance Comparison

Feature	HashSet	LinkedHashSet	TreeSet
Duplicates Allowed?	❌ No	❌ No	❌ No
Order Maintained?	❌ No	✅ Yes (Insertion Order)	✅ Yes (Sorted Order)
Null Allowed?	✅ Yes (One `null`)	✅ Yes (One `null`)	❌ No
Performance	✅ Fastest `O(1)`	✅ `O(1)`	❌ Slower `O(log n)`

7 When to Use Which Set?

Use Case	Best Choice
Fastest insertions, deletions, lookups	✅ `HashSet`
Maintain insertion order	✅ `LinkedHashSet`
Store elements in sorted order	✅ `TreeSet`

Would you like a **real-world project using `Set` in Java?** 🚀

Heading : 24

Subheading : 18

Content : 16