Collection Framework –

The **Java Collection Framework (JCF)** is a set of classes and interfaces in the java.util package that provides data structures and algorithms for storing, processing, and manipulating data efficiently.

Key Interfaces in the Collection Framework

1. Collection Interface

- The root interface for most JCF classes.
- Methods: add(), remove(), size(), clear(), iterator(), etc.

2. List Interface

- Ordered collection that allows duplicate elements.
- Implementations:
  - **ArrayList**: Resizable array. Fast for random access.
  - **LinkedList**: Doubly linked list. Efficient for insertions/deletions.
  - **Vector**: Synchronized ArrayList (legacy).

3. Set Interface

- A collection that does not allow duplicate elements.
- Implementations:
  - **HashSet**: Backed by a hash table. Unordered.
  - **LinkedHashSet**: Maintains insertion order.
  - **TreeSet**: Sorted set based on a Red-Black tree.

4. Queue Interface

- Represents a collection of elements in a FIFO (First In, First Out) order.
- Implementations:
  - **PriorityQueue**: Elements are ordered based on priority.
  - **Deque**: Double-ended queue. Examples: ArrayDeque.

5. Map Interface

- Represents key-value pairs. Does not extend Collection.
- Implementations:
  - **HashMap**: Unordered, allows one null key and multiple null values.
  - **LinkedHashMap**: Maintains insertion order.

- **TreeMap**: Sorted by keys (natural order or custom comparator).
- **Hashtable**: Synchronized (legacy).

Arraylist

The **ArrayList** is a part of the **Java Collection Framework** and resides in the java.util package. It provides a **dynamic array** that can grow or shrink in size as elements are added or removed. It is one of the most commonly used data structures in Java due to its simplicity and efficiency.

Constructors

1. **Default Constructor**

ArrayList<E> list = new ArrayList<>();

Creates an empty list with an initial capacity of 10.

2. **Parameterized Constructor**

ArrayList<E> list = new ArrayList<>(int initialCapacity);

Creates a list with a specified initial capacity.

Method	**Description**
add(E e)	Adds an element to the list.
add(int index, E e)	Inserts an element at a specific index.
remove(int index)	Removes the element at the specified index.

remove(Object o)	Removes the first occurrence of the specified object.
get(int index)	Returns the element at the specified index.
set(int index, E e)	Replaces the element at the specified index.
size()	Returns the number of elements in the list.
clear()	Removes all elements from the list.
contains(Object o)	Checks if the list contains a specific element.
isEmpty()	Checks if the list is empty.
indexOf(Object o)	Returns the first occurrence index of an element.
lastIndexOf(Object o)	Returns the last occurrence index of an element.
subList(int from, int to)	Returns a portion of the list between two indices.

```java
import java.util.ArrayList;

public class ArrayListTutorial {
    public static void main(String[] args) {
        // Step 1: Create an ArrayList
        ArrayList<String> fruits = new ArrayList<>();

        // Step 2: Add elements to the ArrayList
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");
        System.out.println("Fruits List After Adding: " + fruits);

        // Step 3: Access an element by index
        String firstFruit = fruits.get(0);
        System.out.println("First Fruit: " + firstFruit);

        // Step 4: Update an element
        fruits.set(1, "Blueberry");
        System.out.println("Fruits List After Updating: " + fruits);

        // Step 5: Remove an element by index
        fruits.remove(2); // Removes "Cherry"
```

```
        System.out.println("Fruits List After Removing by Index: " + fruits);

        // Step 6: Remove an element by value
        fruits.remove("Blueberry");
        System.out.println("Fruits List After Removing by Value: " + fruits);

        // Step 7: Check if an element exists
        boolean hasApple = fruits.contains("Apple");
        System.out.println("Contains 'Apple': " + hasApple);

        // Step 8: Get the size of the ArrayList
        int size = fruits.size();
        System.out.println("Size of Fruits List: " + size);

        // Step 9: Iterate through the ArrayList
        System.out.println("Iterating through the ArrayList:");
        for (String fruit : fruits) {
            System.out.println(fruit);
        }

        // Step 10: Clear the ArrayList
        fruits.clear();
        System.out.println("Fruits List After Clearing: " + fruits);

        // Step 11: Check if ArrayList is empty
        boolean isEmpty = fruits.isEmpty();
        System.out.println("Is Fruits List Empty? " + isEmpty);
    }
}
```

Output :

```
Fruits List After Adding: [Apple, Banana, Cherry]
First Fruit: Apple
Fruits List After Updating: [Apple, Blueberry, Cherry]
Fruits List After Removing by Index: [Apple, Blueberry]
Fruits List After Removing by Value: [Apple]
Contains 'Apple': true
Size of Fruits List: 1
Iterating through the ArrayList:
Apple
Fruits List After Clearing: []
Is Fruits List Empty? true
```

LinkedList

The LinkedList class in Java is a part of the Java Collection Framework and resides in the java.util package. It provides a doubly linked list implementation, allowing efficient insertion and deletion of elements at both ends or in the middle of the list.

Constructors

1. Default Constructor

LinkedList<E> list = new LinkedList<>();

Creates an empty linked list.

2. Collection Constructor

LinkedList<E> list = new LinkedList<>(Collection<? extends E> c);

Creates a linked list containing the elements of the specified collection.

Method	Description
add(E e)	Appends the element to the end of the list.
add(int index, E element)	Inserts the element at the specified index.
addFirst(E e)	Adds the element at the beginning of the list.
addLast(E e)	Adds the element at the end of the list.
remove()	Removes the first element.
remove(int index)	Removes the element at the specified index.

Method	Description
removeFirst()	Removes the first element.
removeLast()	Removes the last element.
get(int index)	Returns the element at the specified index.
getFirst()	Retrieves the first element.
getLast()	Retrieves the last element.
set(int index, E element)	Replaces the element at the specified index.
size()	Returns the number of elements in the list.
clear()	Removes all elements from the list.
contains(Object o)	Checks if the list contains a specific element.
isEmpty()	Checks if the list is empty.
iterator()	Returns an iterator for the list.
offer(E e)	Adds an element to the end of the list (used in queues).
peek()	Retrieves the head element without removing it.
poll()	Retrieves and removes the head element.

Code Example

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {
        // Step 1: Create a LinkedList
        LinkedList<String> list = new LinkedList<>();

        // Step 2: Add elements
        list.add("Apple");
```

```
        list.add("Banana");
        list.add("Cherry");
        System.out.println("Initial List: " + list);

        // Step 3: Add elements at specific positions
        list.addFirst("Mango");
        list.addLast("Orange");
        System.out.println("After Adding First and Last: " + list);

        // Step 4: Access elements
        System.out.println("First Element: " + list.getFirst());
        System.out.println("Last Element: " + list.getLast());

        // Step 5: Remove elements
        list.removeFirst(); // Removes "Mango"
        list.removeLast();  // Removes "Orange"
        System.out.println("After Removing First and Last: " + list);

        // Step 6: Check for elements
        boolean hasBanana = list.contains("Banana");
        System.out.println("Contains 'Banana': " + hasBanana);

        // Step 7: Iterate through the LinkedList
        System.out.println("Iterating through the List:");
        for (String fruit : list) {
            System.out.println(fruit);
        }

        // Step 8: Clear the LinkedList
        list.clear();
        System.out.println("List After Clearing: " + list);

        // Step 9: Check if empty
        System.out.println("Is List Empty? " + list.isEmpty());
    }
}
```

Output
mathematica
Copy code

```
Initial List: [Apple, Banana, Cherry]
After Adding First and Last: [Mango, Apple, Banana, Cherry, Orange]
First Element: Mango
Last Element: Orange
After Removing First and Last: [Apple, Banana, Cherry]
Contains 'Banana': true
Iterating through the List:
```

```
Apple
Banana
Cherry
List After Clearing: []
Is List Empty? true
```

Stack

A Stack is a linear data structure that follows the LIFO (Last In, First Out) principle. This means the last element added to the stack is the first one to be removed. It is commonly used for tasks such as parsing, backtracking, and evaluating expressions.

Method	Description
push(E item)	Adds an element to the top of the stack.
pop()	Removes and returns the top element of the stack.
peek()	Returns the top element without removing it.
isEmpty()	Checks if the stack is empty.
search(Object o)	Returns the 1-based position of an element in the stack.
size()	Returns the number of elements in the stack.
clear()	Removes all elements from the stack.

Code Example

```
import java.util.Stack;

public class StackExample {
    public static void main(String[] args) {
        // Step 1: Create a Stack
        Stack<Integer> stack = new Stack<>();

        // Step 2: Push elements onto the stack
        stack.push(10);
        stack.push(20);
        stack.push(30);
        System.out.println("Stack after pushes: " + stack);

        // Step 3: Peek at the top element
        System.out.println("Top Element (Peek): " + stack.peek());

        // Step 4: Pop an element from the stack
        int poppedElement = stack.pop();
        System.out.println("Popped Element: " + poppedElement);
        System.out.println("Stack after pop: " + stack);

        // Step 5: Check if the stack is empty
        System.out.println("Is Stack Empty? " + stack.isEmpty());

        // Step 6: Search for an element
        int position = stack.search(10);
        System.out.println("Position of 10 in Stack: " + position);

        // Step 7: Clear the stack
        stack.clear();
        System.out.println("Stack after clearing: " + stack);
    }
}
```

```
Output
mathematica
Copy code
Stack after pushes: [10, 20, 30]
Top Element (Peek): 30
Popped Element: 30
Stack after pop: [10, 20]
Is Stack Empty? false
Position of 10 in Stack: 2
Stack after clearing: []
```

Queue

A Queue is a linear data structure that follows the FIFO (First In, First Out) principle. In a queue, the element added first is processed first, making it suitable for situations like task scheduling, buffering, or managing resources.

Method	Description
add(E e)	Inserts the specified element into the queue; throws exception if it fails.
offer(E e)	Inserts the element into the queue; returns false if it fails.
remove()	Removes and returns the head of the queue; throws exception if queue is empty.
poll()	Removes and returns the head of the queue; returns null if queue is empty.
element()	Retrieves the head without removing it; throws exception if queue is empty.
peek()	Retrieves the head without removing it; returns null if queue is empty.
isEmpty()	Checks if the queue is empty.
size()	Returns the number of elements in the queue.
clear()	Removes all elements from the queue.

Code Example: Basic Queue Operations

```
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args) {
        // Step 1: Create a Queue
        Queue<String> queue = new LinkedList<>();
```

```
        // Step 2: Add elements to the queue
        queue.add("A");
        queue.add("B");
        queue.add("C");
        System.out.println("Queue after additions: " + queue);

        // Step 3: Access the head of the queue
        System.out.println("Head of the queue (Peek): " + queue.peek());

        // Step 4: Remove elements from the queue
        System.out.println("Removed element: " + queue.poll());
        System.out.println("Queue after removal: " + queue);

        // Step 5: Check the size of the queue
        System.out.println("Queue size: " + queue.size());

        // Step 6: Check if the queue is empty
        System.out.println("Is the queue empty? " + queue.isEmpty());

        // Step 7: Clear the queue
        queue.clear();
        System.out.println("Queue after clearing: " + queue);
    }
}
```

Output
yaml
Copy code

```
Queue after additions: [A, B, C]
Head of the queue (Peek): A
Removed element: A
Queue after removal: [B, C]
Queue size: 2
Is the queue empty? false
Queue after clearing: []
```

Types of Queues

1. Simple Queue:
    - Follows FIFO; addition at the rear, removal from the front.
2. Circular Queue:
    - Last position connects back to the first to form a circle.

    - Solves the limitation of static queue size.
3. Priority Queue:
    - Elements are dequeued based on priority, not FIFO order.
4. Double-Ended Queue (Deque):
    - Elements can be added or removed from both ends.

Queue Implementations

Using LinkedList

A linked list implementation allows for dynamic resizing.

```
Queue<Integer> queue = new LinkedList<>();
queue.add(1);
queue.add(2);
queue.poll(); // Removes 1
```

Using ArrayDeque

Provides better performance than LinkedList.

```
Queue<Integer> queue = new ArrayDeque<>();
queue.offer(5);
queue.offer(10);
queue.poll(); // Removes 5
```

Using PriorityQueue

Orders elements based on natural ordering or a custom comparator.

```
import java.util.PriorityQueue;
```

```
PriorityQueue<Integer> priorityQueue = new PriorityQueue<>();
priorityQueue.add(10);
priorityQueue.add(5);
priorityQueue.add(20);
System.out.println(priorityQueue.poll()); // Removes 5 (smallest)
```

SET

A **Set** in Java is a part of the **Java Collection Framework** that represents a collection of unique elements. It does not allow duplicate elements and is typically used when you want to prevent redundancy in a collection.

Key Features of Set

1. **No Duplicates**: Each element in a set is unique.
2. **Unordered Collection**: Most implementations do not maintain insertion order.
3. **Efficient Lookup**: Search operations are faster in sets, especially in **HashSet**.
4. **Implements Collection Interface**: Part of the Java Collection Framework.

Set Hierarchy

Common Implementations

1. **HashSet**:
    - Backed by a hash table.
    - Does not maintain order.
    - Best for frequent add, remove, and contains operations.

```
Set<Integer> set = new HashSet<>();
```

2. **LinkedHashSet**:
    - Maintains insertion order.
    - Slower than HashSet due to linked structure.

```
Set<Integer> set = new LinkedHashSet<>();
```

3. **TreeSet**:
    - Implements a **NavigableSet** backed by a Red-Black tree.
    - Maintains elements in sorted order.

```
Set<Integer> set = new TreeSet<>();
```

Key Methods of Set Interface

Method	Description
add(E e)	Adds the specified element to the set if it is not already present.
remove(Object o)	Removes the specified element from the set, if present.
contains(Object o)	Checks if the set contains the specified element.
isEmpty()	Checks if the set is empty.
size()	Returns the number of elements in the set.
clear()	Removes all the elements from the set.
iterator()	Returns an iterator over the elements in the set.

Code Example: HashSet

```java
import java.util.HashSet;
import java.util.Set;

public class HashSetExample {
    public static void main(String[] args) {
        // Step 1: Create a HashSet
        Set<String> set = new HashSet<>();

        // Step 2: Add elements
        set.add("Apple");
        set.add("Banana");
        set.add("Cherry");
        set.add("Apple"); // Duplicate, will be ignored
        System.out.println("Set after adding elements: " + set);

        // Step 3: Check if an element exists
        System.out.println("Contains 'Banana': " + set.contains("Banana"));

        // Step 4: Remove an element
        set.remove("Banana");
        System.out.println("Set after removing 'Banana': " + set);

        // Step 5: Iterate through the set
        System.out.println("Iterating over Set:");
        for (String fruit : set) {
            System.out.println(fruit);
        }

        // Step 6: Clear the set
        set.clear();
        System.out.println("Is the set empty? " + set.isEmpty());
    }
}
```

Output

```sql
Copy code
Set after adding elements: [Apple, Cherry, Banana]
Contains 'Banana': true
Set after removing 'Banana': [Apple, Cherry]
Iterating over Set:
Apple
Cherry
Is the set empty? true
```

TreeSet Example

```
import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {
        // Step 1: Create a TreeSet
        TreeSet<Integer> treeSet = new TreeSet<>();

        // Step 2: Add elements
        treeSet.add(50);
        treeSet.add(10);
        treeSet.add(20);
        treeSet.add(40);
        System.out.println("TreeSet: " + treeSet); // Sorted order

        // Step 3: Access first and last elements
        System.out.println("First Element: " + treeSet.first());
        System.out.println("Last Element: " + treeSet.last());

        // Step 4: Remove an element
        treeSet.remove(20);
        System.out.println("TreeSet after removal: " + treeSet);

        // Step 5: Iterate over the TreeSet
        System.out.println("Iterating over TreeSet:");
        for (Integer num : treeSet) {
            System.out.println(num);
        }
    }
}
```

Output
yaml
Copy code

```
TreeSet: [10, 20, 40, 50]
First Element: 10
Last Element: 50
TreeSet after removal: [10, 40, 50]
Iterating over TreeSet:
10
40
50
```

MAP

A **Map** is a part of the **Java Collection Framework** used to store **key-value pairs**, where each key maps to exactly one value. It is useful for quickly accessing data based on unique keys.

Key Features of Map

1. **Key-Value Pairs**: Data is stored as pairs, with unique keys and associated values.
2. **No Duplicate Keys**: Each key must be unique, but values can be duplicated.
3. **Efficient Lookup**: Allows fast retrieval of values using keys.
4. **Part of Java Collection Framework**: Includes multiple implementations like HashMap, TreeMap, and LinkedHashMap.

Common Implementations

1. **HashMap**:
    - Backed by a hash table.
    - Does not maintain any order of keys.
    - Allows one null key and multiple null values.

Map<Integer, String> map = new HashMap<>();

2. **LinkedHashMap**:
    - Maintains the insertion order of keys.
    - Slightly slower than HashMap due to its linked structure.

Map<Integer, String> map = new LinkedHashMap<>();

3. **TreeMap**:

- Implements a Red-Black tree.
- Keys are maintained in sorted (natural or custom) order.
- Does not allow null keys.

```
Map<Integer, String> map = new TreeMap<>();
```

4. **Hashtable**:
    - Synchronized (thread-safe) implementation of Map.
    - Does not allow null keys or values.

```
Map<Integer, String> map = new Hashtable<>();
```

Key Methods of Map Interface

Method	Description
put(K key, V value)	Associates the specified key with the specified value.
get(Object key)	Returns the value associated with the specified key.
remove(Object key)	Removes the key-value pair for the specified key.
containsKey(Object key)	Checks if the map contains the specified key.
containsValue(Object v)	Checks if the map contains the specified value.
keySet()	Returns a set view of all the keys in the map.
values()	Returns a collection view of all the values in the map.
entrySet()	Returns a set view of all key-value pairs (entries) in the map.
size()	Returns the number of key-value pairs in the map.

Method	Description
clear()	Removes all the key-value pairs from the map.

Code Example: HashMap

```
import java.util.HashMap;
import java.util.Map;

public class HashMapExample {
    public static void main(String[] args) {
        // Step 1: Create a HashMap
        Map<Integer, String> map = new HashMap<>();

        // Step 2: Add key-value pairs
        map.put(1, "Apple");
        map.put(2, "Banana");
        map.put(3, "Cherry");
        System.out.println("Initial Map: " + map);

        // Step 3: Access a value using its key
        System.out.println("Value for key 2: " + map.get(2));

        // Step 4: Remove a key-value pair
        map.remove(3);
        System.out.println("Map after removal: " + map);

        // Step 5: Check for a key or value
        System.out.println("Contains key 1: " + map.containsKey(1));
        System.out.println("Contains value 'Cherry': " +
map.containsValue("Cherry"));

        // Step 6: Iterate over the map
        System.out.println("Iterating over Map:");
        for (Map.Entry<Integer, String> entry : map.entrySet()) {
            System.out.println("Key: " + entry.getKey() + ", Value: " +
entry.getValue());
        }
    }
}
```

Output
yaml
Copy code
Initial Map: {1=Apple, 2=Banana, 3=Cherry}

```
Value for key 2: Banana
Map after removal: {1=Apple, 2=Banana}
Contains key 1: true
Contains value 'Cherry': false
Iterating over Map:
Key: 1, Value: Apple
Key: 2, Value: Banana
```

TreeMap Example

```java
import java.util.TreeMap;

public class TreeMapExample {
    public static void main(String[] args) {
        // Step 1: Create a TreeMap
        TreeMap<Integer, String> treeMap = new TreeMap<>();

        // Step 2: Add key-value pairs
        treeMap.put(30, "Thirty");
        treeMap.put(10, "Ten");
        treeMap.put(20, "Twenty");
        System.out.println("TreeMap: " + treeMap); // Sorted by keys

        // Step 3: Access first and last entries
        System.out.println("First Entry: " + treeMap.firstEntry());
        System.out.println("Last Entry: " + treeMap.lastEntry());

        // Step 4: Remove an entry
        treeMap.remove(20);
        System.out.println("TreeMap after removal: " + treeMap);
    }
}
```

```
Output
mathematica
Copy code
TreeMap: {10=Ten, 20=Twenty, 30=Thirty}
First Entry: 10=Ten
Last Entry: 30=Thirty
TreeMap after removal: {10=Ten, 30=Thirty}
```