**Java Collections Framework**

**Introduction to Java Collections Framework**

**Objective:**

The primary objective of this document is to provide an in-depth understanding of the Java Collections Framework, its core components, and why it is essential for Java developers to master it. The Collections Framework offers a standard way of handling groups of objects in Java, providing powerful data structures and algorithms for managing data efficiently.



**What is the Collections Framework?**

**Definition and Importance**

The Java Collections Framework (JCF) is a unified architecture for representing and manipulating collections of objects. It provides a set of interfaces, implementations, and algorithms to efficiently store and process data. The JCF is fundamental for working with data structures in Java, as it allows developers to easily perform operations like searching, sorting, inserting, updating, and removing elements from data structures.

The importance of the Collections Framework lies in its ability to provide reusable data structures, saving developers from the need to build their own data structures from scratch. By using pre-built classes and interfaces, developers can improve productivity, reduce errors, and write more efficient code.

**Core Interfaces in the Framework**

The Collections Framework defines several core interfaces that specify different types of collections. These interfaces define the fundamental operations and behaviors that different data structures must support.

* **Collection Interface:** The root of the collection hierarchy.
* **List Interface:** Ordered collection that allows duplicates and provides indexed access.
* **Set Interface:** Unordered collection with no duplicate elements.
* **Queue Interface:** A collection designed for holding elements prior to processing, typically following FIFO (First-In-First-Out) principle.
* **Deque Interface:** A double-ended queue that supports operations on both ends.
* **Map Interface:** A collection of key-value pairs, where each key is unique.

**Core Implementations**

The core interfaces of the Java Collections Framework are implemented by various classes. These implementations are optimized for different use cases, providing various trade-offs in terms of time complexity, memory usage, and operation speed.

The core interfaces in the Java Collections Framework are implemented by various classes, each offering unique characteristics and performance trade-offs. These implementations provide developers with a range of options, making it easier to choose the most appropriate one for their specific use cases.

**1. List Interface Implementations**

* **ArrayList**
  + **Description**: Resizable array implementation of the List interface. It allows fast random access to elements and is ideal for scenarios where you mostly read from the collection, but infrequently insert or delete elements.
  + **Key Characteristics**:
    - Supports dynamic resizing.
    - Fast access to elements via index.
    - Slower insertion/deletion in the middle of the list.
  + **Time Complexity**:
    - Get: O(1)
    - Add (at the end): O(1)
    - Add (at an arbitrary position): O(n)
    - Remove (at an arbitrary position): O(n)

import java.util.ArrayList;

import java.util.List;

public class ArrayListExample {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

// 1. add() - Adds an element at the end of the list

list.add("Apple");

list.add("Banana");

list.add("Cherry");

// 2. add(index, element) - Adds an element at a specific index

list.add(1, "Mango");

// 3. get() - Retrieves an element by index

System.out.println("Element at index 2: " + list.get(2));

// 4. remove() - Removes the element at a specific index

list.remove(1); // Removes "Mango"

// 5. contains() - Checks if the list contains a specific element

System.out.println("List contains 'Banana': " + list.contains("Banana"));

// Printing the final list

System.out.println("Final List: " + list);

}

}

* **LinkedList**
  + **Description**: Doubly linked list implementation of the List interface. Allows for fast insertion and removal of elements, especially at the beginning or end of the list.
  + **Key Characteristics**:
    - Ideal for applications that require frequent insertions or deletions.
    - Slower access to elements by index (due to traversal).
    - Also implements the Deque and Queue interfaces.
  + **Time Complexity**:
    - Get: O(n)
    - Add (at the beginning or end): O(1)
    - Add (at an arbitrary position): O(n)
    - Remove (at an arbitrary position): O(n)

import java.util.LinkedList;

public class LinkedListExample {

public static void main(String[] args) {

LinkedList<String> linkedList = new LinkedList<>();

// 1. add() - Adds an element to the end of the list

linkedList.add("Apple");

linkedList.add("Banana");

linkedList.add("Orange");

// 2. addFirst() - Adds an element to the front of the list

linkedList.addFirst("Grapes");

// 3. addLast() - Adds an element to the end of the list

linkedList.addLast("Pineapple");

// 4. removeFirst() - Removes the first element of the list

System.out.println("Removed first: " + linkedList.removeFirst());

// 5. removeLast() - Removes the last element of the list

System.out.println("Removed last: " + linkedList.removeLast());

// Printing the final list

System.out.println("Final LinkedList: " + linkedList);

}

}

* **Vector**
  + **Description**: An older implementation of the List interface, similar to ArrayList, but synchronized.
  + **Key Characteristics**:
    - Synchronized methods make it thread-safe, but performance can be slower than ArrayList.
    - Legacy class.
  + **Time Complexity**:
    - Same as ArrayList, but with added synchronization overhead.

import java.util.Vector;

public class VectorExample {

public static void main(String[] args) {

Vector<String> vector = new Vector<>();

// 1. add() - Adds an element to the end of the vector

vector.add("Red");

vector.add("Green");

vector.add("Blue");

// 2. insertElementAt() - Adds an element at a specific position

vector.insertElementAt("Yellow", 1); // Inserts "Yellow" at index 1

// 3. remove() - Removes the element at a specific index

vector.remove(2); // Removes "Blue" from index 2

// 4. elementAt() - Retrieves the element at a specific index

System.out.println("Element at index 1: " + vector.elementAt(1));

// 5. contains() - Checks if a specific element exists in the vector

System.out.println("Vector contains 'Red': " + vector.contains("Red"));

// Printing the final vector

System.out.println("Final Vector: " + vector);

}

}

* **Stack**
  + **Description**: Extends Vector and implements the Stack class with LIFO (Last In, First Out) semantics.
  + **Key Characteristics**:
    - Provides methods like push(), pop(), and peek() for stack-based operations.
    - Considered a legacy class, with Deque (specifically ArrayDeque) recommended for modern stack-like behavior.
  + **Time Complexity**:
    - Push/Pop: O(1)

import java.util.Stack;

public class StackExample {

public static void main(String[] args) {

Stack<String> stack = new Stack<>();

// 1. push() - Pushes an element onto the stack

stack.push("One");

stack.push("Two");

stack.push("Three");

// 2. pop() - Removes and returns the top element of the stack

System.out.println("Popped: " + stack.pop()); // Removes "Three"

// 3. peek() - Returns, but does not remove, the top element of the stack

System.out.println("Peeked: " + stack.peek()); // Peeks at "Two"

// 4. search() - Returns the 1-based position of the element from the top of the stack

System.out.println("Position of 'One': " + stack.search("One"));

// 5. empty() - Checks if the stack is empty

System.out.println("Is the stack empty? " + stack.empty());

// Printing the final stack

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

}

}

**2. Set Interface Implementations**

* **HashSet**
  + **Description**: Implements the Set interface using a hash table. It does not maintain any order of elements.
  + **Key Characteristics**:
    - Guarantees that no duplicate elements are stored.
    - Provides constant-time performance for add, remove, and contains operations, assuming a good hash function.
  + **Time Complexity**:
    - Add: O(1)
    - Contains: O(1)
    - Remove: O(1)

import java.util.HashSet;

import java.util.Set;

public class HashSetExample {

public static void main(String[] args) {

Set<String> set = new HashSet<>();

// 1. add() - Adds an element to the set

set.add("Red");

set.add("Blue");

set.add("Green");

// 2. add() - Does not allow duplicate elements

set.add("Blue"); // Duplicate will be ignored

// 3. remove() - Removes an element from the set

set.remove("Green");

// 4. contains() - Checks if the set contains a specific element

System.out.println("Set contains 'Blue': " + set.contains("Blue"));

// 5. size() - Returns the size of the set

System.out.println("Size of set: " + set.size());

// Printing the final set

System.out.println("Final Set: " + set);

}

}

* **LinkedHashSet**
  + **Description**: Extends HashSet, but maintains the insertion order of elements (i.e., elements are ordered by the order in which they were added).
  + **Key Characteristics**:
    - Slightly slower than HashSet due to maintaining the order.
    - Useful when you need to preserve insertion order.
  + **Time Complexity**:
    - Same as HashSet, but with additional overhead for maintaining the order.

import java.util.LinkedHashSet;

public class LinkedHashSetExample {

public static void main(String[] args) {

LinkedHashSet<String> linkedHashSet = new LinkedHashSet<>();

// 1. add() - Adds an element to the set

linkedHashSet.add("Apple");

linkedHashSet.add("Banana");

linkedHashSet.add("Orange");

// 2. add() - Does not allow duplicate elements

linkedHashSet.add("Banana"); // Duplicate will be ignored

// 3. remove() - Removes a specific element

linkedHashSet.remove("Orange");

// 4. contains() - Checks if the set contains a specific element

System.out.println("Set contains 'Apple': " + linkedHashSet.contains("Apple"));

// 5. size() - Returns the size of the set

System.out.println("Size of set: " + linkedHashSet.size());

// Printing the final set

System.out.println("Final LinkedHashSet: " + linkedHashSet);

}

}

* **TreeSet**
  + **Description**: Implements the Set interface using a Red-Black tree. Elements are stored in a sorted order, according to their natural ordering or a provided comparator.
  + **Key Characteristics**:
    - Guarantees that elements are sorted.
    - Provides logarithmic time performance for add, remove, and contains operations.
  + **Time Complexity**:
    - Add/Remove/Contains: O(log n)

import java.util.TreeSet;

public class TreeSetExample {

public static void main(String[] args) {

TreeSet<Integer> treeSet = new TreeSet<>();

// 1. add() - Adds an element to the set

treeSet.add(10);

treeSet.add(20);

treeSet.add(15);

// 2. remove() - Removes a specific element

treeSet.remove(15);

// 3. first() - Retrieves the first (smallest) element in the set

System.out.println("First element: " + treeSet.first());

// 4. last() - Retrieves the last (largest) element in the set

System.out.println("Last element: " + treeSet.last());

// 5. size() - Returns the size of the set

System.out.println("Size of TreeSet: " + treeSet.size());

// Printing the final TreeSet

System.out.println("Final TreeSet: " + treeSet);

}

}

**3. Queue Interface Implementations**

* **PriorityQueue**
  + **Description**: Implements the Queue interface and provides a priority-based ordering of elements. Elements are ordered according to their natural ordering or a specified comparator.
  + **Key Characteristics**:
    - Often used in algorithms requiring elements to be processed based on priority (e.g., Dijkstra's algorithm, task scheduling).
    - Does not guarantee FIFO ordering, only priority ordering.
  + **Time Complexity**:
    - Add: O(log n)
    - Remove (poll/peek): O(log n)

import java.util.PriorityQueue;

import java.util.Queue;

public class PriorityQueueExample {

public static void main(String[] args) {

Queue<Integer> queue = new PriorityQueue<>();

// 1. add() - Adds an element to the queue

queue.add(10);

queue.add(30);

queue.add(20);

// 2. peek() - Retrieves, but does not remove, the head of the queue

System.out.println("Head of the queue: " + queue.peek());

// 3. poll() - Retrieves and removes the head of the queue

System.out.println("Polled element: " + queue.poll());

// 4. offer() - Adds an element to the queue (returns false if full)

queue.offer(5);

// 5. size() - Returns the number of elements in the queue

System.out.println("Size of the queue: " + queue.size());

// Printing the final queue

System.out.println("Final Queue: " + queue);

}

}

* **LinkedList** (also implements Queue and Deque)
  + **Description**: Can be used as both a Queue (FIFO) and Deque (LIFO).
  + **Key Characteristics**:
    - Allows efficient insertion and deletion of elements at both ends of the list.
  + **Time Complexity**:
    - Add/Remove (at both ends): O(1)

**4. Deque Interface Implementations**

* **ArrayDeque**
  + **Description**: Resizable array implementation of the Deque interface. It supports efficient insertion and removal at both ends.
  + **Key Characteristics**:
    - Ideal for applications that require stack or queue functionality.
    - Does not allow null elements.
  + **Time Complexity**:
    - Add/Remove (at both ends): O(1)

import java.util.ArrayDeque;

import java.util.Deque;

public class ArrayDequeExample {

public static void main(String[] args) {

Deque<String> deque = new ArrayDeque<>();

// 1. addFirst() - Adds an element to the front of the deque

deque.addFirst("One");

// 2. addLast() - Adds an element to the end of the deque

deque.addLast("Two");

// 3. removeFirst() - Removes and returns the first element

System.out.println("Removed from front: " + deque.removeFirst());

// 4. removeLast() - Removes and returns the last element

deque.addLast("Three");

System.out.println("Removed from end: " + deque.removeLast());

// 5. peekFirst() - Retrieves, but does not remove, the first element

deque.addFirst("Four");

System.out.println("First element: " + deque.peekFirst());

// Printing the final deque

System.out.println("Final Deque: " + deque);

}

}

* **LinkedList** (also implements Deque)
  + **Description**: In addition to being a List, LinkedList implements the Deque interface and allows insertion and removal of elements from both ends of the list.
  + **Key Characteristics**:
    - Useful when you need to treat the collection as both a stack (LIFO) and a queue (FIFO).
  + **Time Complexity**:
    - Add/Remove (at both ends): O(1)

import java.util.LinkedList;

import java.util.Queue;

import java.util.Deque;

public class LinkedListQueueDequeExample {

public static void main(String[] args) {

// Using LinkedList as both Queue and Deque

Deque<String> deque = new LinkedList<>();

// Queue methods

System.out.println("Queue Methods:");

// 1. add() - Adds an element to the end of the queue

deque.add("Element1");

deque.add("Element2");

deque.add("Element3");

// 2. peek() - Retrieves, but does not remove, the head of the queue

System.out.println("Peeked element: " + deque.peek());

// 3. poll() - Retrieves and removes the head of the queue

System.out.println("Polled element: " + deque.poll());

// 4. size() - Returns the number of elements in the queue

System.out.println("Queue size: " + deque.size());

// 5. isEmpty() - Checks if the queue is empty

System.out.println("Is the queue empty? " + deque.isEmpty());

// Deque methods

System.out.println("\nDeque Methods:");

// 6. addFirst() - Adds an element to the front of the deque

deque.addFirst("FirstElement");

// 7. addLast() - Adds an element to the end of the deque

deque.addLast("LastElement");

// 8. peekFirst() - Retrieves, but does not remove, the first element

System.out.println("First element: " + deque.peekFirst());

// 9. peekLast() - Retrieves, but does not remove, the last element

System.out.println("Last element: " + deque.peekLast());

// 10. removeFirst() - Removes and returns the first element

System.out.println("Removed first element: " + deque.removeFirst());

// 11. removeLast() - Removes and returns the last element

System.out.println("Removed last element: " + deque.removeLast());

// 12. offerFirst() - Inserts the specified element at the front of the deque

deque.offerFirst("OfferedFirst");

// 13. offerLast() - Inserts the specified element at the end of the deque

deque.offerLast("OfferedLast");

// 14. pollFirst() - Retrieves and removes the first element of the deque

System.out.println("Polled first element: " + deque.pollFirst());

// 15. pollLast() - Retrieves and removes the last element of the deque

System.out.println("Polled last element: " + deque.pollLast());

// Printing the final deque

System.out.println("\nFinal Deque: " + deque);

}

}

**5. Map Interface Implementations**

* **HashMap**
  + **Description**: Implements the Map interface using a hash table. Stores key-value pairs, and does not allow duplicate keys.
  + **Key Characteristics**:
    - Offers constant-time performance for get, put, and remove operations, assuming a good hash function.
    - Does not guarantee any specific order for keys or values.
  + **Time Complexity**:
    - Get/Put/Remove: O(1) (average case)

import java.util.HashMap;

import java.util.Map;

public class HashMapExample {

public static void main(String[] args) {

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

// 1. put() - Adds key-value pair to the map

map.put("Apple", 50);

map.put("Banana", 30);

map.put("Orange", 20);

// 2. get() - Retrieves the value associated with a specific key

System.out.println("Price of Banana: " + map.get("Banana"));

// 3. remove() - Removes the key-value pair associated with a specific key

map.remove("Orange");

// 4. containsKey() - Checks if the map contains a specific key

System.out.println("Map contains key 'Apple': " + map.containsKey("Apple"));

// 5. size() - Returns the number of key-value pairs in the map

System.out.println("Size of map: " + map.size());

// Printing the final map

System.out.println("Final Map: " + map);

}

}

* **LinkedHashMap**
  + **Description**: Extends HashMap and maintains the order of insertion for the key-value pairs.
  + **Key Characteristics**:
    - Slightly slower than HashMap due to order maintenance, but still offers fast performance.
    - Ideal when insertion order is required.
  + **Time Complexity**:
    - Same as HashMap, with added overhead for maintaining the order.

import java.util.LinkedHashMap;

import java.util.Map;

public class LinkedHashMapExample {

public static void main(String[] args) {

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

// 1. put() - Adds key-value pair to the map

map.put("1", "One");

map.put("2", "Two");

map.put("3", "Three");

// 2. get() - Retrieves the value associated with a specific key

System.out.println("Value for key '2': " + map.get("2"));

// 3. remove() - Removes the key-value pair associated with a specific key

map.remove("3");

// 4. keySet() - Returns a set view of the keys in the map

System.out.println("Keys: " + map.keySet());

// 5. values() - Returns a collection view of the values in the map

System.out.println("Values: " + map.values());

// Printing the final map

System.out.println("Final LinkedHashMap: " + map);

}

}

* **TreeMap**
  + **Description**: Implements the Map interface using a Red-Black tree, which stores key-value pairs in a sorted order according to the natural ordering of the keys or a custom comparator.
  + **Key Characteristics**:
    - Provides a sorted map.
    - Slower than HashMap for basic operations due to the need to maintain order.
  + **Time Complexity**:
    - Get/Put/Remove: O(log n)

import java.util.TreeMap;

public class TreeMapExample {

public static void main(String[] args) {

TreeMap<String, Integer> treeMap = new TreeMap<>();

// 1. put() - Adds key-value pairs to the map

treeMap.put("Apple", 50);

treeMap.put("Banana", 30);

treeMap.put("Orange", 20);

// 2. get() - Retrieves the value associated with a specific key

System.out.println("Price of Banana: " + treeMap.get("Banana"));

// 3. remove() - Removes a specific key-value pair

treeMap.remove("Orange");

// 4. firstKey() - Retrieves the first (lowest) key

System.out.println("First key: " + treeMap.firstKey());

// 5. lastKey() - Retrieves the last (highest) key

System.out.println("Last key: " + treeMap.lastKey());

// Printing the final TreeMap

System.out.println("Final TreeMap: " + treeMap);

}

}

* **Hashtable** (Legacy)
  + **Description**: An older, synchronized implementation of the Map interface. It is similar to HashMap, but is thread-safe and has been mostly replaced by ConcurrentHashMap.
  + **Key Characteristics**:
    - Synchronized methods for thread safety.
    - Slow performance compared to HashMap.
  + **Time Complexity**:
    - Same as HashMap, with synchronization overhead.
* **ConcurrentHashMap**
  + **Description**: A thread-safe implementation of the Map interface, designed for high concurrency. It allows multiple threads to read and write concurrently without blocking each other.
  + **Key Characteristics**:
    - Unlike Hashtable, ConcurrentHashMap allows concurrent read access and segregates the map into segments to improve concurrency.
  + **Time Complexity**:
    - Get: O(1) (in most cases)
    - Put: O(1) (in most cases)

import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample {

public static void main(String[] args) throws InterruptedException {

// Create a ConcurrentHashMap

ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

// Thread 1: Adds elements to the map

Thread thread1 = new Thread(() -> {

map.put("Apple", 10);

map.put("Banana", 20);

map.put("Orange", 30);

System.out.println("Thread 1 added elements: " + map);

});

// Thread 2: Adds and modifies elements in the map

Thread thread2 = new Thread(() -> {

map.put("Apple", 15); // Modify value of "Apple"

map.put("Grapes", 25);

map.put("Mango", 35);

System.out.println("Thread 2 added and modified elements: " + map);

});

// Thread 3: Removes elements from the map

Thread thread3 = new Thread(() -> {

map.remove("Banana");

map.remove("Mango"); // This entry was added by thread2

System.out.println("Thread 3 removed elements: " + map);

});

// Start all threads

thread1.start();

thread2.start();

thread3.start();

// Wait for all threads to finish

thread1.join();

thread2.join();

thread3.join();

// Final state of the map

System.out.println("Final ConcurrentHashMap: " + map);

// Additional Operations on ConcurrentHashMap

// 1. get() - Retrieve a value associated with a key

System.out.println("Value for 'Apple': " + map.get("Apple"));

// 2. containsKey() - Check if a key exists

System.out.println("Does 'Banana' exist in the map? " + map.containsKey("Banana"));

// 3. size() - Get the size of the map

System.out.println("Size of map: " + map.size());

}

}

**Utility Classes**

**1. Collections**

The Collections class provides static methods for working with collections. It offers various utility methods like sorting, reversing, and finding the maximum or minimum element in a collection. Examples of common methods include:

* sort()
* reverse()
* shuffle()
* max() / min()

import java.util.\*;

public class CollectionsExample {

public static void main(String[] args) {

// Create a List of integers

List<Integer> numbers = new ArrayList<>(Arrays.asList(5, 2, 8, 1, 6, 3, 7, 4));

// 1. sort() - Sort the list in natural (ascending) order

System.out.println("Original List: " + numbers);

Collections.sort(numbers);

System.out.println("Sorted List (Ascending): " + numbers);

// 2. reverse() - Reverse the order of elements in the list

Collections.reverse(numbers);

System.out.println("Reversed List: " + numbers);

// 3. shuffle() - Shuffle the elements randomly

Collections.shuffle(numbers);

System.out.println("Shuffled List: " + numbers);

// 4. max() - Find the maximum element in the list

Integer maxElement = Collections.max(numbers);

System.out.println("Maximum Element: " + maxElement);

// 5. min() - Find the minimum element in the list

Integer minElement = Collections.min(numbers);

System.out.println("Minimum Element: " + minElement);

// Using a custom comparator to demonstrate max() / min() with a custom condition

List<String> names = new ArrayList<>(Arrays.asList("John", "Alice", "Bob", "Charlie", "David"));

// Using max() with a custom comparator (compare by string length)

String longestName = Collections.max(names, Comparator.comparingInt(String::length));

System.out.println("Longest Name: " + longestName);

// Using min() with a custom comparator (compare by string length)

String shortestName = Collections.min(names, Comparator.comparingInt(String::length));

System.out.println("Shortest Name: " + shortestName);

}

}

**Iterators and How They Work**

An iterator is an object that enables traversal over the elements in a collection. It provides methods like next(), hasNext(), and remove(). Iterators are essential for iterating over various collection types without exposing the underlying data structure.

Example:

List<String> list = new ArrayList<>();

list.add("A");

list.add("B");

list.add("C");

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()) {

System.out.println(iterator.next());

}