UNIT IV – JAVA COLLECTIONS FRAMEWORK

4.1. Java Collections Framework:

- The Java Collection Framework is a unified architecture introduced in JDK 1.2 for storing, retrieving, and manipulating groups of objects.
- It provides a set of interfaces and classes to handle data structures like lists, sets, maps, and queues efficiently.
- All these are part of the java.util package.S

Key Features of JCF:

- Provides standardized interfaces for data structures.
- Supports dynamic size (unlike arrays).
- Implements data manipulation algorithms (sorting, searching, shuffling).
- Offers type-safety via Generics.
- Supports iteration through Iterator and ListIterator.
- Ensures high performance with efficient implementations.

Key Components of the Java Collection Framework

1. **Interfaces**: Define the abstract data types for collections.
   - **List**: Ordered collection (e.g., ArrayList, LinkedList).
   - **Set**: Unordered collection with no duplicate elements (e.g., HashSet, TreeSet).
   - **Map**: Key-value pairs (e.g., HashMap, TreeMap).
   - **Queue**: FIFO (First-In-First-Out) structure (e.g., PriorityQueue, LinkedList).
2. **Classes**: Concrete implementations of the interfaces.
   - **ArrayList**: Resizable array implementation of the List interface.
   - **HashSet**: Implements the Set interface using a hash table.
   - **HashMap**: Implements the Map interface using a hash table.
   - **LinkedList**: Implements both List and Queue interfaces.
3. **Algorithms**: Utility methods provided by the Collections class for operations like sorting, searching, and shuffling.

Advantages of the Java Collection Framework

- **Reusability**: Predefined classes and interfaces reduce the need for custom implementations.
- **Efficiency**: Optimized data structures improve performance.
- **Flexibility**: Supports various types of collections (ordered, unordered, etc.).
- **Thread Safety**: Some classes like Vector and Hashtable are synchronized.

4.1.1. List:

- List is an interface within the java.util package that represents an ordered collection of elements.
- It extends the Collection interface and allows for duplicate elements and null values (depending on the specific implementation).

Key characteristics and features of a Java List:

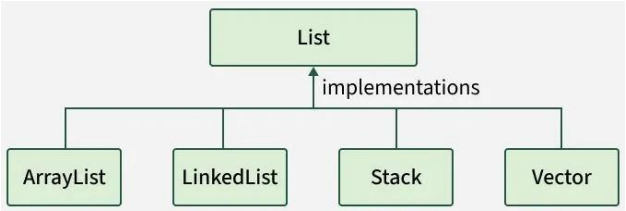
- **Ordered Collection:** Elements are maintained in the order they were

inserted, allowing for index-based access and manipulation.

- **Allows Duplicates:** Unlike Sets, Lists can contain multiple instances of the same element.
- **Index-based Operations:** Provides methods for positional access, such as get(), set(), add(), remove() at a specific index.

Implementations:

Since List is an interface, it cannot be directly instantiated.



Common implementing classes include:

- ✓ ArrayList: A resizable array implementation, efficient for random access but potentially slower for insertions/deletions in the middle.
- ✓ LinkedList: A doubly-linked list implementation, efficient for insertions/deletions at the beginning or end, but slower for random access.
- ✓ Vector: A legacy, synchronized version of ArrayList.
- ✓ Stack: A LIFO (Last-In, First-Out) data structure that extends Vector.

4.1.1.1 : ArrayList:

- ➢ An ArrayList in Java is a resizable array implementation of the List interface, found in the java.util package.
- ➢ It provides a dynamic array that can grow or shrink in size as elements are added or removed, offering more flexibility compared to traditional fixed-size Java arrays.

Key characteristics of ArrayLists:

Resizable Array: Unlike standard arrays, you do not need to specify a fixed size at the time of creation. The ArrayList automatically manages its capacity as elements are added or removed.

Indexed Access: Elements in an ArrayList can be accessed using their integer index, similar to arrays, providing constant-time (O(1)) retrieval using the get() method.

Generics Support: ArrayLists support generics, allowing you to specify the type of objects they will store, ensuring type safety at compile-time. For example, ArrayList<String> will only store String objects.

Not Synchronized: ArrayLists are not inherently thread-safe. If multiple threads access and modify an ArrayList concurrently, external synchronization mechanisms (like Collections.synchronizedList()) are required.

Allows Null and Duplicates: ArrayLists can store both null values and duplicate elements.

Maintains Insertion Order: Elements are stored in the order in which they are added to the ArrayList.

Stores Objects: ArrayLists are designed to store objects. To store primitive data types (like int, double, char), their corresponding wrapper classes (e.g., Integer, Double, Character) must be used.

Common ArrayList operations:

- ✓ **Creation:** ArrayList<Type> listName = new ArrayList<>();
- ✓ **Adding Elements:** listName.add(element);

- ✓ **Accessing Elements:** listName.get(index);
- ✓ **Removing Elements:** listName.remove(index); or listName.remove(object);
- ✓ **Checking Size:** listName.size();
- ✓ **Iterating:** Using a for loop with get() and size(), or using a for-each loop.

Example:

```
import java.util.ArrayList;
public class ArrayListExample
{
    public static void main(String[] args)
    {
        // Create an ArrayList of Strings
        ArrayList<String> fruits = new ArrayList<>();
        // Add elements
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Orange");
        // Access an element
        System.out.println("First fruit: " + fruits.get(0)); // Output: Apple
        // Remove an element
        fruits.remove("Banana");
        // Iterate and print all elements
        System.out.println("Fruits in the list:");
        for (String fruit : fruits)
        {
            System.out.println(fruit);
        }
    }
}
```

```
D:\Java>java  ArrayListExample
First fruit: Apple
Fruits in the list:
Apple
Orange
```

4.1.1.2 : LinkedList:

- A LinkedList is a part of the Java Collections Framework and implements the List and Deque interfaces.
- It uses a doubly linked list internally to store its elements.

Key characteristics of LinkedList in Java:

- **Dynamic Size:** Unlike arrays, LinkedList can grow or shrink dynamically as elements are added or removed, without needing to pre-define its capacity.
- **Non-contiguous Memory:** Elements in a LinkedList are not stored in contiguous memory locations. Instead, each element (node) contains the data and references (links) to the previous and next nodes.
- **Efficient Insertions and Deletions:** Adding or removing elements in a LinkedList is generally more efficient than in an ArrayList, especially at the beginning or middle of the list, as it only requires updating a few pointers.
- **Slower Random Access:** Accessing elements by index (e.g., get(index)) is less efficient than in an ArrayList because it requires traversing the list from the beginning or end until the desired index is reached.

Implementation of List and Deque:

This means LinkedList can be used as a standard list, a queue (First-In, First-Out), or a stack (Last-In, First-Out) due to its Deque interface implementation.

Example:

```
import java.util.LinkedList;
public class LinkedListExample
```

```
{
    public static void main(String[] args)
    {
        // Create an LinkedList of Strings
        LinkedList<String> fruits = new LinkedList<>();
        // Add elements
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Orange");
        // Access an element
        System.out.println("First fruit: " + fruits.get(0)); // Output: Apple
        // Remove an element
        fruits.remove("Banana");
        // Iterate and print all elements
        System.out.println("Fruits in the list:");
        for (String fruit : fruits)
      {
            System.out.println(fruit);
        }
    }
}
```

```
D:\Java>java  LinkedListExample
First fruit: Apple
Fruits in the list:
Apple
Orange
```

ArrayList vs LinkedList:

ArrayList	LinkedList
Underlying structure is Dynamic Array.	Underlying structure is Doubly-linked list.
O(1) - Fast random access.	O(n) - Slow random access.
Memory is lower (contiguous memory).	Memory is higher (extra pointers per node).
Iteration speed is faster.	Iteration speed is slower.
Insertion and deletion is slower.	Insertion and deletion is faster.

4.1.2. Set:

- The Set interface is a part of the Java Collection Framework, located in the java.util package.
- It represents a collection of unique elements, meaning it does not allow duplicate values.

Key Features of Set

- **No duplicates:** Set does not allow duplicate elements; each item must be unique.
- **No guaranteed order:** Elements in a Set are not stored or retrieved in any defined order.

- **Ordering exceptions:** LinkedHashSet maintains insertion order, while TreeSet keeps elements sorted.
- **Collection methods inherited:** Set supports standard methods like add(), remove(), contains(), size() and iterator() from the Collection interface.

- **One null allowed:** Most Set implementations allow only a single null element.

Common Implementations of the Set Interface:

- **HashSet:** This is the most commonly used Set implementation. It uses a hash table for storage, offering fast performance for basic operations like add(), remove(), and contains(). It does not guarantee any iteration order.
- **LinkedHashSet:** This implementation maintains the insertion order of elements, meaning elements are iterated in the order they were added. It also offers near-constant time performance for basic operations.
- **TreeSet:** This implementation stores elements in a sorted order, either naturally (if elements implement Comparable) or according to a custom Comparator. It provides guaranteed logarithmic time performance for basic operations.

4.1.2.1 : HashSet:

- A HashSet in Java is a class that implements the Set interface, providing a collection for storing unique elements.
- It utilizes a hash table (specifically, a HashMap internally) for efficient storage and retrieval of elements.

Key Characteristics of HashSet:

Uniqueness:

- ✓ It ensures that only unique elements are stored. Attempting to add a duplicate element will not result in its addition, and the add() method will return false.

No Order Guarantee:

- ✓ Elements are not stored in any specific order (like insertion order or sorted order). The iteration order is not guaranteed and can vary.
- ✓ Allows one null element:
- ✓ A HashSet can contain a single null element. Adding multiple null values will only store one.

Performance:

- ✓ Due to its hash table implementation, HashSet offers efficient (average O(1) time complexity) operations for adding, removing, and checking for the presence of elements.

Non-Synchronized:

- ✓ HashSet is not thread-safe.
- ✓ If used in a multi-threaded environment where multiple threads might modify the set concurrently, external synchronization is required.

Example:

```
import java.util.HashSet;
public class HashSetExample
{
   public static void main(String[] args)
   {
      // Create an HashSet of Strings
      HashSet<String> fruits = new HashSet<>();
      // Add elements
```

```
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Orange");
fruits.add("Apple");
```

```
        // Remove an element
        fruits.remove("Banana");
        // Iterate and print all elements
        System.out.println("Fruits in the list:");
        for (String fruit : fruits)
        {
            System.out.println(fruit);
        }
    }
}
```

```
D:\Java>java  HashSetExample
Fruits in the list:
Apple
Orange
```

4.1.2.2 : TreeSet:

- TreeSet in Java is a part of the Java Collections Framework and implements the SortedSet and NavigableSet interfaces.
- It stores unique elements in a sorted order, either according to their natural ordering or by a Comparator provided at the time of creation.

Key Characteristics of TreeSet:

Sorted Order:

- ✓ Elements are automatically stored and retrieved in ascending order (natural ordering) or based on a custom Comparator.

Unique Elements:

- ✓ Like all Set implementations, TreeSet does not allow duplicate elements.
- ✓ Attempting to add a duplicate will not result in an error but will simply not add the element.

Underlying Data Structure:

- ✓ Internally, TreeSet uses a TreeMap to store its elements, specifically a self-balancing binary search tree (like a Red-Black Tree) for efficient operations.

Performance:

- ✓ Operations like insertion, deletion, and searching have a time complexity of O(log n), where 'n' is the number of elements in the set.

No Null Elements (with natural ordering):

- ✓ If relying on natural ordering, TreeSet generally does not accept null elements, as compareTo() would throw a NullPointerException.
- ✓ If a custom Comparator is provided that handles nulls, then nulls can potentially be stored.

Not Thread-Safe:

- ✓ TreeSet is not synchronized.
- ✓ If multiple threads access a TreeSet concurrently and at least one of the threads modifies the set, it must be synchronized externally.

Example:

```
import java.util.TreeSet;
public class TreeSetExample
{
    public static void main(String[] args)
    {
```

```
// Create an TreeSet of Strings
TreeSet<String> fruits = new TreeSet<>();
// Add elements
fruits.add("Apple");
```

```
        fruits.add("Banana");
        fruits.add("Orange");
        fruits.add("Apple");
        // Remove an element
        fruits.remove("Banana");
        // Iterate and print all elements
        System.out.println("Fruits in the list:");
        for (String fruit : fruits)
        {
            System.out.println(fruit);
        }
    }
}
```

```
D:\Java>java  TreeSetExample
Fruits in the list:
Apple
Orange
```

HashSet Vs TreeSet:

S. No.	HashSet	TreeSet
1	Hash set is implemented using HashTable	The tree set is implemented using a tree structure.
2	HashSet allows a null object	The tree set does not allow the null object. It throws the null pointer exception.
3	Hash set use equals method to compare two objects	Tree set use compare method for comparing two objects.
4	Hash set doesn't now allow a heterogeneous object	Tree set allows a heterogeneous object
5	HashSet does not maintain any order	TreeSet maintains an object in sorted order

4.1.3. Map:

- In Java, a Map is an interface within the java.util package that represents a collection of key-value pairs.
- Each key in a Map must be unique, and it maps to a specific value. Values, however, can be duplicated.

Key Characteristics of Java Maps:

Key-Value Association: Maps store data as pairs, where a unique key is associated with a corresponding value.

Unique Keys: No two keys in a single Map instance can be identical.

Duplicate Values Allowed: Multiple keys can map to the same value.

No Direct Extension of Collection Interface: While part of the Java Collections Framework, Map does not directly extend the Collection interface.

Common Implementations of the Map Interface:

✓ **HashMap:** Provides fast, unordered storage. It's generally the most efficient for

general-purpose use cases where insertion order or sorted order is not required.

- ✓ **TreeMap:** Stores entries in a sorted order based on the natural ordering of the keys or by a custom Comparator.

- ✓ **LinkedHashMap:** Maintains the insertion order of elements, meaning elements are retrieved in the order they were added.
- ✓ **ConcurrentHashMap:** A thread-safe implementation suitable for concurrent environments, offering high performance for concurrent operations.

Basic Map Operations and Methods:

- ✓ **put(K key, V value):** Adds a new key-value pair or updates the value for an existing key.
- ✓ **get(Object key):** Retrieves the value associated with the specified key.
- ✓ **remove(Object key):** Removes the key-value pair associated with the specified key.
- ✓ **containsKey(Object key):** Checks if the map contains the specified key.
- ✓ **containsValue(Object value):** 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 the key-value mappings (entries) in the map.

Example:

```
import java.util.HashMap;
public class HashMapExample
{
    public static void main(String[] args)
    {
        // Create a HashMap with Integer as key and String as value
        HashMap<Integer, String> fruits = new HashMap<>();
        // Add elements
        fruits.put(1, "Apple");
        fruits.put(2, "Banana");
        fruits.put(3, "Orange");
        fruits.put(4, "Apple");  // duplicates allowed with different keys
        System.out.println("Fruits in the list:");
        for (Integer key : fruits.keySet())
        {
            System.out.println(key + " -> " + fruits.get(key));
        }
        // Remove element by key
        fruits.remove(2);
        // Iterate and print all elements
        System.out.println("After Removing Banana..");
        for (Integer key : fruits.keySet())
        {
            System.out.println(key + " -> " + fruits.get(key));
        }
    }
}
```

```
D:\Java>java  HashMapExample
Fruits in the list:
1 -> Apple
2 -> Banana
3 -> Orange
4 -> Apple
After Removing Banana..
1 -> Apple
3 -> Orange
4 -> Apple
```