Class-13 : Java Collections Framework

by Pial Kanti Samadder

What is Collections Framework

- A *collection* is an object that represents a group of objects.
- A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.
- The Java Collections Framework provides a set of **interfaces** and a set of **classes** that implement those interfaces.
- All of these are part of the `java.util` package.
- They are used to store, search, sort, and organize data more easily - all using standardized methods and patterns.

Core Interfaces in the Collections Framework

Here are some common interfaces, along with their classes:

Interface	Common Classes	Description
`List`	`ArrayList`, `LinkedList`	Ordered collection that allows duplicates
`Set`	`HashSet`, `TreeSet`, `LinkedHashSet`	Collection of unique elements
`Map`	`HashMap`, `TreeMap`, `LinkedHashMap`	Stores key-value pairs with unique keys

How it simplifies data handling

1. **Reduces programming effort -** by providing data structures and algorithms so you don't have to write them yourself.
2. **Increases performance -** by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be tuned by switching implementations.
3. **Provides interoperability between unrelated APIs -** by establishing a common language to pass collections back and forth.
4. **Reduces the effort required to learn APIs -** by requiring you to learn multiple ad hoc collection APIs.
5. **Reduces the effort required to design and implement APIs -** by not requiring you to produce ad hoc collections APIs.
6. **Fosters software reuse -** by providing a standard interface for collections and algorithms with which to manipulate them.

Traversing Collections

Three ways to traverse a collection:

1. **Aggregate Operations (Streams)**
    - Modern, concise, expressive
2. **For-each Construct**
    - Simple and readable loop
3. **Iterators**
    - Fine control, safe removal

Aggregate Operations (Streams)

- Introduced in **JDK 8**
- Use **Streams + Lambda Expressions**
- **Non-mutative** (don’t change original collection)
- Can be **parallelized** for performance

```
myShapes.stream()
   .filter(s -> s.getColor() == Color.RED)
   .forEach(s -> System.out.println(s.getName()));
```

For-each Construct

- Short and **easy-to-read** loop
- Best for simple iteration
- **Cannot remove** elements inside loop

```
for (Object o : collection) {
    System.out.println(o);
}
```

Iterators

- An **Iterator** lets you step through elements one by one
- Methods:
  - `hasNext()` → checks if more elements
  - `next()` → returns next element
  - `remove()` → safely removes last element

```
for (Iterator<String> it = list.iterator(); it.hasNext();) {
    if (it.next().isEmpty()) {
        it.remove(); // safe removal
    }
}
```

Collection Interface Bulk Operations

Bulk operations perform an operation on an entire `Collection`. They are usually more efficient than manually coding the same actions.

The following are the bulk operations:

- `containsAll` — returns `true` if the target `Collection` contains all of the elements in the specified `Collection`.
- `addAll` — adds all of the elements in the specified `Collection` to the target `Collection`.
- `removeAll` — removes from the target `Collection` all of its elements that are also contained in the specified `Collection`.
- `retainAll` — removes from the target `Collection` all its elements that are *not* also contained in the specified `Collection`. That is, it retains only those elements in the target `Collection` that are also contained in the specified `Collection`.
- `clear` — removes all elements from the `Collection`.

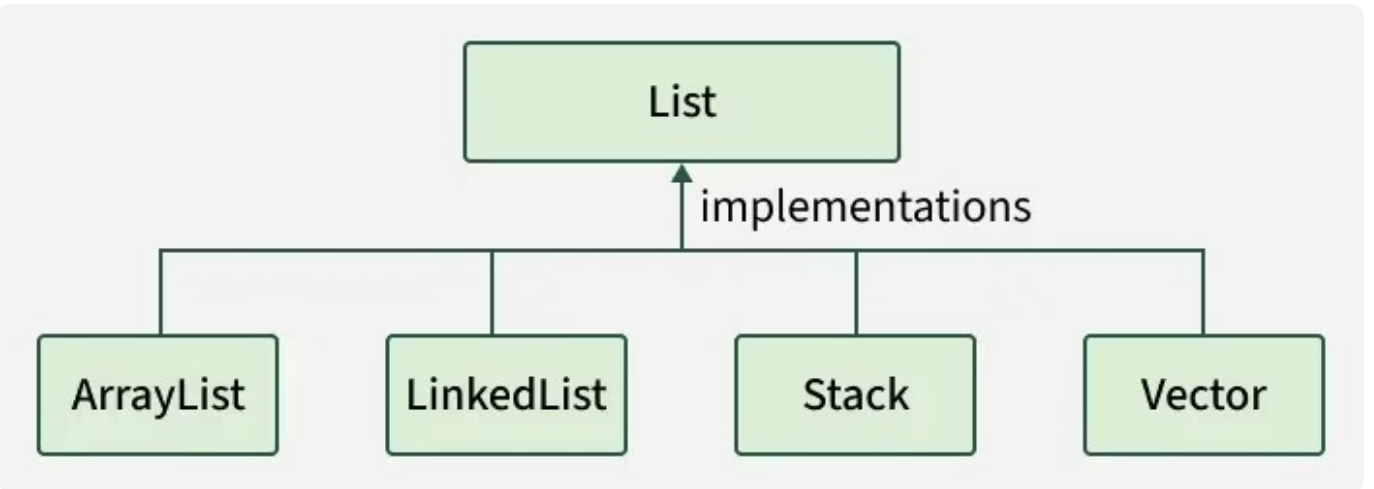
List Interface

The **List** interface in Java is a fundamental component of the Java Collections Framework, designed to represent an ordered collection of elements.

Key Characteristics Of The List Interface

1. **Ordered Collection**: Lists preserve the order of elements, ensuring that elements are stored in the sequence in which they were added.
2. **Duplicates Allowed**: Lists permit duplicate elements. Multiple occurrences of the same element can coexist in a list.
3. **Indexed Access**: Elements can be retrieved and modified using their index positions.
4. **Dynamic Size**: Lists can dynamically grow or shrink as elements are added or removed.

Common implementation classes of the List interface



1. **ArrayList:** It implement using resizable array, offers fast random access but slower insert/delete.
2. **LinkedList:** It implement using Doubly-linked list, efficient for frequent insertions and deletions.
3. **Vector:** It implement using Legacy synchronized dynamic array, thread-safe but slower.
4. **Stack:** It implement using a LIFO (Last-In-First-Out) subclass of Vector for stack operations.

ArrayList

- In Java, an ArrayList is a resizable array implementation that is part of the java.util package.
- Unlike regular arrays, you don’t need to specify its size in advance; it can grow or shrink dynamically as elements are added or removed.

Key Features

- **Resizable Array:** ArrayList can automatically grow dynamically in size.
- **Indexed Access:** ArrayList elements can be accessed using indices like arrays.
- **Supports Generics:** It ensures type safety at compile-time.
- **Not Synchronized:** ArrayList uses Collections.synchronizedList() for thread safety.
- **Allows Null and Duplicates:** ArrayList allows both null values and duplicate elements.
- **Maintains Insertion Order:** Elements are stored in the order they are added.

Example

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        // Creating an ArrayList of Strings
        ArrayList<String> fruits = new ArrayList<>();

        // Add elements
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Mango");
        System.out.println("After adding elements: " + fruits);

        // Insert element at specific index
        fruits.add(1, "Orange"); // Insert "Orange" at index 1
        System.out.println("After inserting Orange at index 1: " + fruits);

        // Access element by index
        System.out.println("Element at index 2: " + fruits.get(2));

        // Remove element by index
        fruits.remove(3); // Remove the element at index 3
        System.out.println("After removing element at index 3: " + fruits);

        // Remove element by value
        fruits.remove("Apple");
        System.out.println("After removing Apple: " + fruits);

        // Iterate over elements
        System.out.println("Iterating over ArrayList:");
        for (String fruit : fruits) {
            System.out.println(fruit);
        }

        // Get size
        System.out.println("Size of ArrayList: " + fruits.size());

        // Allow null
        fruits.add(null);
        System.out.println("After adding null: " + fruits);
    }
}
```

LinkedList

- LinkedList is a part of the Java Collection Framework and is present in the java.util package.
- It implements a doubly-linked list data structure where elements (called nodes) are not stored in contiguous memory.
- Each node contains two parts: data and a reference to the next (and previous) node.

Insert an Element at a Specific Position in a *LinkedList*

Method	Description
addFirst(E e)	Adds an element at the beginning of a list
addLast(E e)	Adds an element at the end of a list
add(E e)	Adds an element at the end of a list
add(int index, E element)	Adds an element at index position *i* of a list

Example

```
import java.util.LinkedList;

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

        // Add elements at the end
        list.add("Banana");       // add(E e)
        list.addLast("Mango");    // addLast(E e)
        System.out.println("After adding Banana and Mango: " + list);

        // Add element at the beginning
        list.addFirst("Apple");   // addFirst(E e)
        System.out.println("After adding Apple at first: " + list);

        // Add element at specific index
        list.add(2, "Orange");    // add(index, element)
        System.out.println("After adding Orange at index 2: " + list);

        // Access elements
        System.out.println("Element at index 1: " + list.get(1));

        // Remove elements
        list.removeFirst();
        list.removeLast();
        System.out.println("After removing first and last: " + list);

        // Iterate over LinkedList
        System.out.println("Iterating over LinkedList:");
        for (String fruit : list) {
            System.out.println(fruit);
        }
    }
}
```

ArrayList vs LinkedList

- **Structure**
  - **ArrayList** → Dynamic array (continuous memory)
  - **LinkedList** → Doubly linked nodes (`prev`, `next`)
- **Access**
  - **ArrayList** → Random access = **O(1)**
  - **LinkedList** → Sequential access = **O(n)**
- **Insertion / Deletion**
  - **ArrayList**
    - End → **Amortized O(1)**
    - Middle → **O(n)** (elements shift)
  - **LinkedList**
    - End → **O(1)**
    - Middle → **O(1)** (if reference known, otherwise O(n))

Example: Insert in ArrayList

Before:

```
[A][B][C][D][E]
```

Insert `X` at index `2` →

```
[A][B][X][C][D][E]
```

Elements `C, D, E` shift → **O(n)**

What does “Amortized” mean?

- Some operations are **cheap most of the time**, but **expensive occasionally** (like resizing ArrayList).
- If you **average the cost** across many operations, it’s still **constant time (O(1))**.
- Example: Adding at end of ArrayList →
  - Usually **O(1)**
  - Occasionally **O(n)** when resizing → spread out cost → **amortized O(1)**

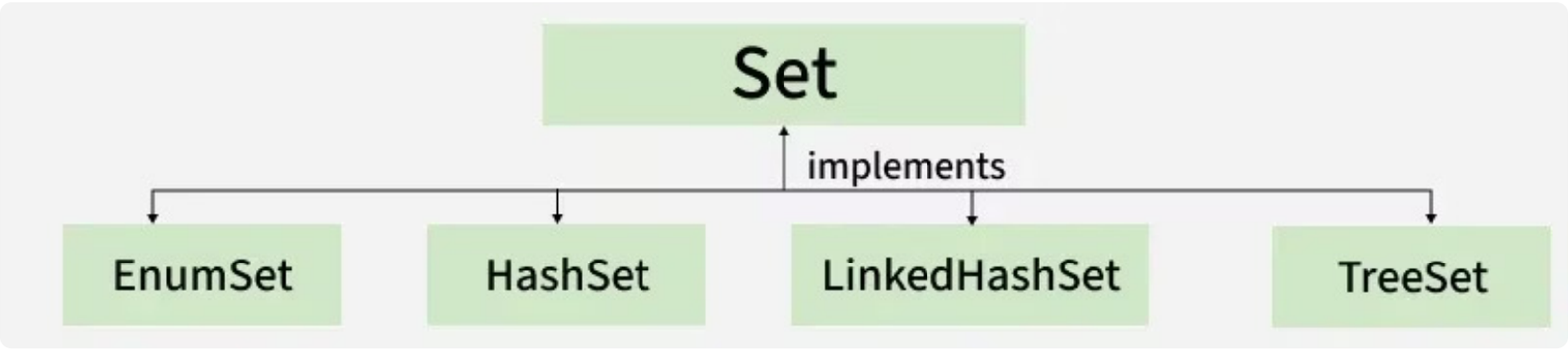
Set Interface

- In Java, 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.
- **One null allowed:** Most Set implementations allow only a single null element.
- **Collection methods inherited:** Set supports standard methods like add(), remove(), contains(), size() and iterator() from the Collection interface.

Hierarchy of Java Set interface



HashSet

- HashSet in Java implements the **Set interface** of Collections Framework.
- It is used to store the unique elements and it doesn't maintain any specific order of elements.
- Can store the Null values.
- Uses **HashMap** (implementation of hash table data structure) internally.
- HashSet is not thread-safe. So to make it thread-safe, synchronization is needed externally.

Example

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

public class HashSetExample {
    public static void main(String[] args) {
        // Create a HashSet of Strings
        Set<String> fruits = new HashSet<>();

        // Add elements
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Mango");
        fruits.add("Orange");

        // Attempt to add duplicate (ignored, no error)
        fruits.add("Apple");

        System.out.println("Fruits: " + fruits);

        // Check if an element exists
        System.out.println("Contains Mango? " + fruits.contains("Mango"));
        System.out.println("Contains Grape? " + fruits.contains("Grape"));

        // Remove an element
        fruits.remove("Banana");
        System.out.println("After removing Banana: " + fruits);

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

        // Iterate over elements
        System.out.println("Iterating over HashSet:");
        for (String fruit : fruits) {
            System.out.println(fruit);
        }

        // Clear all elements
        fruits.clear();
        System.out.println("Is set empty after clear? " + fruits.isEmpty());
    }
}
```

LinkedHashSet

- `LinkedHashSet` combines the functionality of a **HashSet** with a LinkedList to maintain the insertion order of elements.
- Stores unique elements only.
- Maintains insertion order.
- Provides faster iteration compared to HashSet.
- Allows null elements.

Example

```
import java.util.LinkedHashSet;
import java.util.Set;

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

        // Add elements
        cities.add("Dhaka");
        cities.add("London");
        cities.add("New York");
        cities.add("Tokyo");

        // Add duplicate (ignored)
        cities.add("Dhaka");

        // Add null
        cities.add(null);

        // Print elements
        System.out.println("Cities: " + cities);

        // Check existence
        System.out.println("Contains Tokyo? " + cities.contains("Tokyo"));

        // Remove element
        cities.remove("London");
        System.out.println("After removing London: " + cities);

        // Iterate (in insertion order)
        System.out.println("Iterating over LinkedHashSet:");
        for (String city : cities) {
            System.out.println(city);
        }
    }
}
```

TreeSet

A TreeSet is a collection class that stores unique elements in a sorted order. It is part of `java.util` package that implements the SortedSet interface, and internally uses a **Red-Black tree** to maintain sorting.

Key Features of `TreeSet`

- `TreeSet` does not allow duplicate elements; duplicate insertions are ignored.
- `TreeSet` does not allow `null` values; inserting `null` throws a `NullPointerException`.
- Implements the `NavigableSet` interface and provides navigation methods like `higher()`, `lower()`, `ceiling()` and `floor()`.
- Not thread-safe; for concurrent access, it must be synchronized using **Collections.synchronizedSet()**.

Example

```
import java.util.TreeSet;

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

        // Add elements
        numbers.add(50);
        numbers.add(10);
        numbers.add(30);
        numbers.add(20);
        numbers.add(40);

        // Add duplicate (ignored)
        numbers.add(30);

        // Print elements (sorted automatically)
        System.out.println("Numbers: " + numbers);

        // NavigableSet methods
        System.out.println("Higher than 25: " + numbers.higher(25));
        System.out.println("Lower than 25: " + numbers.lower(25));
        System.out.println("Ceiling of 25: " + numbers.ceiling(25));
        System.out.println("Floor of 25: " + numbers.floor(25));
    }
}
```

Map Interface

What is a Map?

- A **Map** stores **key-value pairs**.
- **Keys are unique**; each key maps to **exactly one value**.
- **Values can be duplicates**.

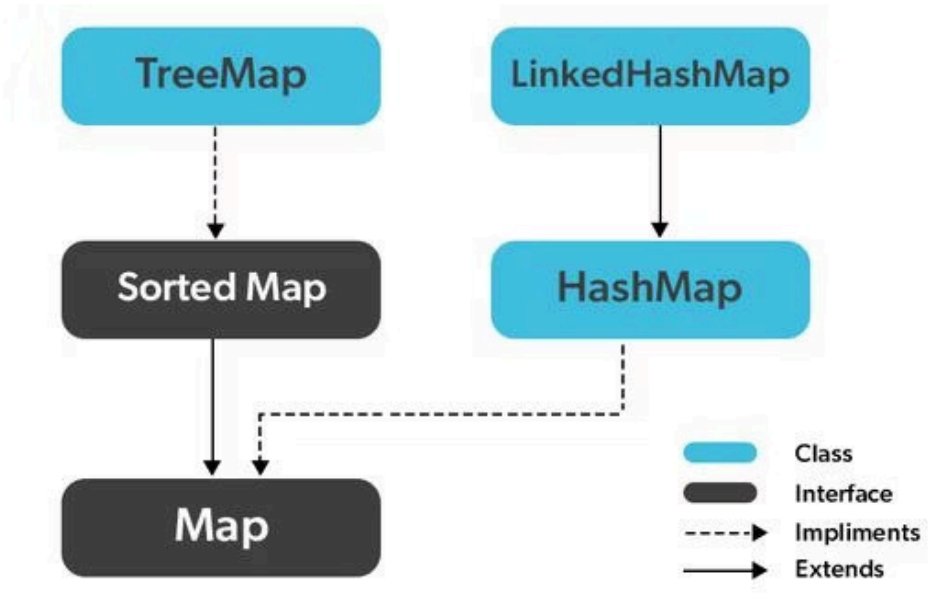
Key Features

- **No duplicate keys** → keys must be unique.
- **Null handling**
  - One `null` key allowed in `HashMap` and `LinkedHashMap`
  - Multiple `null` values allowed
- **Thread-safety**
  - `ConcurrentHashMap` → thread-safe alternative
  - `Collections.synchronizedMap()` → wrapper for synchronization

Map Interfaces & Implementations

Interface	Description	Common Implementations
Map	Basic key-value collection	`HashMap`, `LinkedHashMap`
SortedMap	Maintains keys in sorted order	`TreeMap`

Hierarchy of Map



HashMap

What is HashMap?

- Implements the **Map interface**
- Stores **key-value pairs**
- **Keys are unique**, **values can be duplicated**
- Uses **hashing** internally for **fast access, insertion, and deletion**

Key Features

- **Not synchronized** → faster than `Hashtable`
- **Allows one null key** and multiple null values
- **Duplicate keys** → new value replaces existing value
- Efficient operations → `get`, `put`, `remove` ≈ **O(1)** average

Example

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

public class HashMapExample {
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<>();

        // Add key-value pairs
        map.put(1, "Apple");
        map.put(2, "Banana");
        map.put(3, "Mango");

        // Add a duplicate key (replaces value)
        map.put(2, "Orange");

        // Add null key and null value
        map.put(null, "NullKey");
        map.put(4, null);

        // Print map
        // Display the contents of the HashMap (order may appear sorted sometimes, but it is unpredictable and not guaranteed)
        System.out.println("HashMap: " + map);

        // Access value by key
        System.out.println("Value for key 2: " + map.get(2));

        // Remove element by key
        map.remove(3);
        System.out.println("After removing key 3: " + map);
    }
}
```

LinkedHashMap

LinkedHashMap in Java implements the **Map** interface of the **Collections Framework**. It stores key-value pairs while maintaining the insertion order of the entries. It maintains the order in which elements are added.

- Stores unique key-value pairs.
- Maintains insertion order.
- Allows one null key and multiple null values.

Example

```
import java.util.LinkedHashMap;
import java.util.Map;

public class LinkedHashMapExample {
    public static void main(String[] args) {
        Map<Integer, String> map = new LinkedHashMap<>();

        // Add key-value pairs
        map.put(3, "Mango");
        map.put(1, "Apple");
        map.put(2, "Banana");

        // Add null key and null value
        map.put(null, "NullKey");
        map.put(4, null);

        // Display map (insertion order preserved)
        System.out.println("LinkedHashMap: " + map);
    }
}
```

TreeMap

- Implements **Map** and **NavigableMap** interfaces
- Stores **key-value pairs in sorted order**
  - By **natural key ordering** (Comparable)
  - Or using a **custom Comparator**
- **No null keys allowed** (throws `NullPointerException`)
- Allows **multiple null values**
- Provides **navigation methods**: `higher()`, `lower()`, `ceiling()`, `floor()`

Example

```
import java.util.TreeMap;

public class TreeMapExample {
    public static void main(String[] args) {
        TreeMap<Integer, String> treeMap = new TreeMap<>();

        // Add key-value pairs
        treeMap.put(3, "Mango");
        treeMap.put(1, "Apple");
        treeMap.put(2, "Banana");

        // Display TreeMap (sorted by keys)
        System.out.println("TreeMap: " + treeMap);

        // NavigableMap methods
        System.out.println("Higher than 1: " + treeMap.higherKey(1));
        System.out.println("Lower than 2: " + treeMap.lowerKey(2));
    }
}
```

Choosing the Right Collection in Java

- **List**

Collection	Key Features	Use Case
ArrayList	Dynamic array, fast random access (O(1)), allows duplicates, allows null, maintains insertion order	Frequent **get** operations, rare insertions in middle
LinkedList	Doubly-linked nodes, fast insert/remove at ends (O(1)), allows duplicates, allows null, maintains insertion order	Frequent **insertions/deletions**, less random access

- **Set**

Collection	Key Features	Use Case
HashSet	Unique elements, no order, allows null	When **uniqueness matters**, order doesn't
LinkedHashSet	Unique elements, maintains insertion order, allows null	When **uniqueness + insertion order** matters
TreeSet	Unique elements, sorted order, no null key	When **uniqueness + sorted order** needed

- **Map**

Collection	Key Features	Use Case
HashMap	Key-value pairs, fast O(1) access, allows one null key	Fast **lookup** by key, order not important
LinkedHashMap	Key-value pairs, maintains insertion order, allows one null key	Predictable **iteration order**
TreeMap	Key-value pairs, sorted by keys, no null keys	**Sorted key access**, range queries, navigation methods

When to Use Map vs Set

Use Set When:

- You need **unique elements** only.
- **Duplicates are not allowed**.
- The focus is on **existence** or **membership** checks.
- Examples:
  - Removing duplicates from a list
  - Storing unique IDs, tags, or names
- Common implementations: `HashSet`, `LinkedHashSet`, `TreeSet`

Use Map When:

- You need **key-value association**.
- Each key is **unique**, but values can be duplicated.
- You want **fast lookup, insertion, and deletion by key**.
- Examples:
  - Storing employee ID → employee name
  - Counting occurrences of words (`Map<String, Integer>`)
- Common implementations: `HashMap`, `LinkedHashMap`, `TreeMap`

Quick Tip

- **Set = Collection of unique items**
- **Map = Collection of unique keys + associated values**