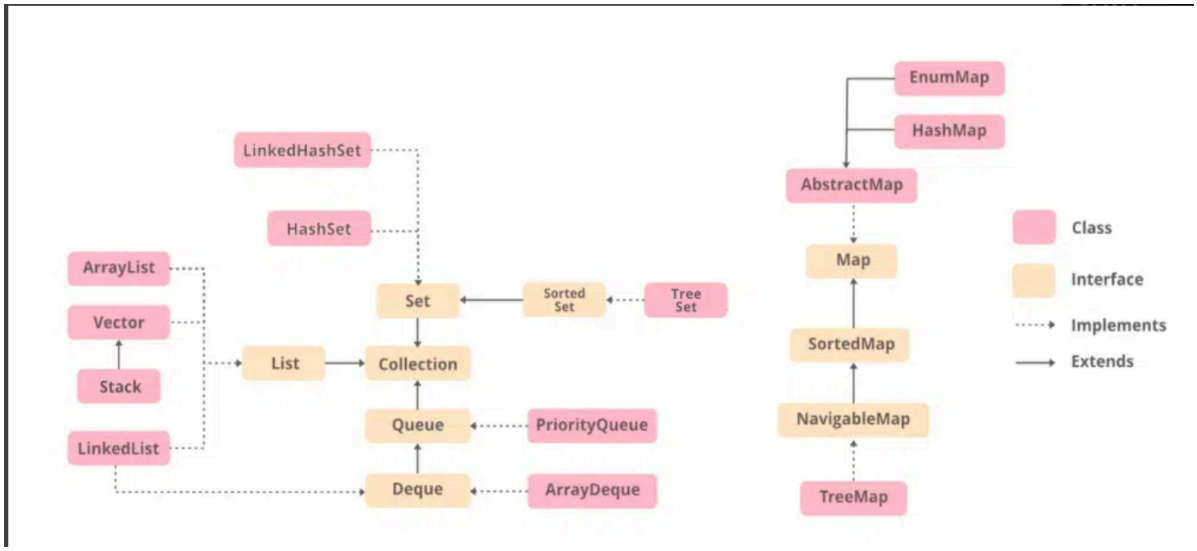
Java Collections Framework



The Java Collections Framework provides a unified architecture for managing and processing groups of objects.

1. Core Interfaces

- **Collection**: The root interface in the framework. It is the super-interface for most of the other collection interfaces like List, Set, and Queue.
    - Common methods:
        - `add(E e)` – Adds an element.
        - `remove(Object o)` – Removes an element.
        - `size()` – Returns the size.
        - `contains(Object o)` – Checks if an element is present.
        - `iterator()` – Returns an iterator for the collection.
        - `clear()` – Removes all elements.
- **List**: An ordered collection (sequence) that allows duplicate elements. Lists provide control over the insertion order.
    - **Common Implementations**:
        - **ArrayList**: Backed by a dynamic array. Fast random access but slow insertions/removals (except at the end).
        - **LinkedList**: Implemented as a doubly linked list. Better performance for insertions and deletions.
        - **Vector**: Similar to ArrayList but synchronized (thread-safe).
        - **Stack**: A subclass of Vector, providing a last-in, first-out (LIFO) stack.
    - **Common Methods** (in addition to Collection):
        - `get(int index)` – Returns the element at the specified index.

        - `add(int index, E element)` – Inserts an element at the specified position.
        - `remove(int index)` – Removes the element at the specified index.
- **Set**: A collection that does not allow duplicate elements. It models the mathematical set.
    - **Common Implementations**:
        - **HashSet**: Uses a hash table, allows constant-time complexity for `add()`, `remove()`, and `contains()`, but does not guarantee order.
        - **LinkedHashSet**: Similar to `HashSet`, but maintains the order in which elements are inserted.
        - **TreeSet**: A NavigableSet implementation backed by a Red-Black tree, which keeps elements sorted according to their natural ordering or a provided comparator.
    - **Common Methods** (in addition to Collection):
        - `add(E e)` – Adds an element, returns false if already exists.
        - `remove(Object o)` – Removes the specified element.
        - `iterator()` – Returns an iterator to traverse the elements.
        - `size()` – Returns the number of elements.
- **Queue**: A collection designed for holding elements before processing. The typical ordering is FIFO (first in, first out).
    - **Common Implementations**:
        - **LinkedList**: A doubly linked list, which can also implement Queue.
        - **PriorityQueue**: A queue where elements are ordered based on priority (uses a comparator or natural ordering).
        - **ArrayDeque**: A resizable array implementation of the `Deque` interface, useful as a double-ended queue.
    - **Common Methods**:
        - `offer(E e)` – Adds an element to the queue.
        - `poll()` – Retrieves and removes the head of the queue.
        - `peek()` – Retrieves but does not remove the head.

2.Map

In Java, a **Map** is an object that maps **keys** to **values**. It is part of the **Java Collections Framework** and is not a subtype of the **Collection** interface (like `Set` and `List`), but it is still a core part of the framework. A **Map** allows storing pairs of key-value data, where each key is unique, and each key maps to exactly one value.

Key Characteristics of Map:

- **Key-Value Pair**: A `Map` stores data as key-value pairs. Each key is associated with a single value.

- **No Duplicate Keys**: A `Map` does not allow duplicate keys. If you try to insert a new key-value pair with an existing key, the old value is replaced by the new one.
- **Unordered**: Some `Map` implementations (like `HashMap`) do not guarantee any specific order of keys or values. Other implementations (like `TreeMap`) maintain a specific order based on natural ordering or a custom comparator.

Core Operations in Map:

- `put(K key, V value)`: Adds a key-value pair to the map.
- `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.
- `size()`: Returns the number of key-value pairs in the map.
- `isEmpty()`: Checks if the map is empty.
- `keySet()`: Returns a `Set` of all keys in the map.
- `values()`: Returns a `Collection` of all values in the map.
- `entrySet()`: Returns a `Set` of all key-value pairs as `Map.Entry` objects.

Common Implementations of Map:

1. **HashMap**:
    - **Order**: Does not guarantee any order of keys or values.
    - **Performance**: Provides constant-time performance for `get` and `put` operations on average.
    - **Usage**: It is the most commonly used `Map` implementation when ordering is not important.

Example:

```
Map<String, Integer> map = new HashMap<>();
map.put("Apple", 3);
map.put("Banana", 5);
map.put("Cherry", 2);

System.out.println(map);  // Output: {Apple=3, Banana=5, Cherry=2}
```

2. **TreeMap**:
    - **Order**: Maintains the keys in **natural order** (for comparable objects) or a custom order defined by a `Comparator`.
    - **Performance**: Offers log(n) time complexity for `get`, `put`, and `remove` operations.

- **Usage**: Useful when you need to maintain the order of keys.

Example:

```
Map<String, Integer> map = new TreeMap<>();
map.put("Apple", 3);
map.put("Banana", 5);
map.put("Cherry", 2);

System.out.println(map);  // Output: {Apple=3, Banana=5, Cherry=2}
```

3. **LinkedHashMap**:
    - **Order**: Maintains the **insertion order** of the keys.
    - **Performance**: Slightly slower than `HashMap` because it maintains the order, but it offers predictable iteration order.
    - **Usage**: Useful when you need to maintain the order of insertion while still benefiting from the constant-time complexity for `get` and `put` operations.

Example:

```
Map<String, Integer> map = new LinkedHashMap<>();
map.put("Apple", 3);
map.put("Banana", 5);
map.put("Cherry", 2);

System.out.println(map);  // Output: {Apple=3, Banana=5, Cherry=2}
```

4. **Hashtable**:
    - **Order**: Like `HashMap`, `Hashtable` does not maintain any order.
    - **Performance**: It is synchronized, which makes it slower than `HashMap` in single-threaded environments.
    - **Usage**: It is largely obsolete, and `HashMap` is generally preferred for most use cases, except for cases where synchronization is needed.

Example:

```
Map<String, Integer> map = new Hashtable<>();
map.put("Apple", 3);
map.put("Banana", 5);
map.put("Cherry", 2);

System.out.println(map);  // Output: {Apple=3, Banana=5, Cherry=2}
```

Example Code of Using Map:

```
import java.util.*;

public class MapExample {
    public static void main(String[] args) {
        // Using HashMap
        Map<String, Integer> hashMap = new HashMap<>();
        hashMap.put("Apple", 3);
        hashMap.put("Banana", 5);
        hashMap.put("Cherry", 2);

        System.out.println("HashMap: " + hashMap);


        // Using TreeMap (Sorted by natural ordering)
        Map<String, Integer> treeMap = new TreeMap<>();
        treeMap.put("Apple", 3);
        treeMap.put("Banana", 5);
        treeMap.put("Cherry", 2);

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

        // Using LinkedHashMap (Insertion order)
        Map<String, Integer> linkedHashMap = new LinkedHashMap<>();
        linkedHashMap.put("Apple", 3);
        linkedHashMap.put("Banana", 5);
        linkedHashMap.put("Cherry", 2);

        System.out.println("LinkedHashMap: " + linkedHashMap);

        // Retrieve value by key
        System.out.println("Value for 'Banana': " +
hashMap.get("Banana"));

        // Remove a key-value pair
        hashMap.remove("Cherry");
        System.out.println("HashMap after removal: " + hashMap);
```

```
        // Iterate through the Map
        System.out.println("Iterating over TreeMap:");
        for (Map.Entry<String, Integer> entry : treeMap.entrySet()) {
            System.out.println(entry.getKey() + ": " +
entry.getValue());
        }
    }
}
```

Output:

yaml
Copy code

```
HashMap: {Apple=3, Banana=5, Cherry=2}
TreeMap: {Apple=3, Banana=5, Cherry=2}
LinkedHashMap: {Apple=3, Banana=5, Cherry=2}
Value for 'Banana': 5
HashMap after removal: {Apple=3, Banana=5}
Iterating over TreeMap:
Apple: 3
Banana: 5
Cherry: 2
```