Java Collections Framework

1. List

Purpose:

- **A `List` is an ordered collection (also known as a sequence) that can contain duplicate elements. The `List` interface provides methods to access elements by their position (index) and to search for elements within the list.**

Key Implementations:

- **`ArrayList`: A resizable array that provides fast random access to elements.**
- **`LinkedList`: A doubly linked list that is faster for inserting or removing elements at the beginning or end.**

Example:

java

```
import java.util.ArrayList;
import java.util.List;

public class ListExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();

        // Adding elements
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        // Accessing elements
        System.out.println("First name: " + names.get(0));
```

```
        // Iterating over the list
        for (String name : names) {
            System.out.println(name);
        }
    }
}
```

2. Set

Purpose:

- **A `Set` is an unordered collection that does not allow duplicate elements. Sets are useful when you need to store unique elements and want to efficiently check for the presence of an element.**

Key Implementations:

- **`HashSet`: Based on a hash table, provides constant-time performance for basic operations (add, remove, contains).**
- **`LinkedHashSet`: Maintains a linked list of the entries in the set, thus preserving the insertion order.**
- **`TreeSet`: A `Set` implementation that keeps its elements in ascending order, based on the natural ordering or a specified comparator.**

Example:

java

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

public class SetExample {
    public static void main(String[] args) {
        Set<String> uniqueNames = new HashSet<>();

        // Adding elements
        uniqueNames.add("Alice");
```

```
        uniqueNames.add("Bob");
        uniqueNames.add("Alice"); // Duplicate, won't be added

        // Checking presence
        System.out.println("Contains 'Alice': " +
uniqueNames.contains("Alice"));

        // Iterating over the set
        for (String name : uniqueNames) {
            System.out.println(name);
        }
    }
}
```

3. Map

Purpose:

- A `Map` is a collection of key-value pairs where each key maps to exactly one value. Keys must be unique, but values can be duplicated.

Key Implementations:

- `HashMap`: Provides constant-time performance for basic operations, uses a hash table.
- `LinkedHashMap`: Like `HashMap`, but maintains the order in which keys were inserted.
- `TreeMap`: A `Map` that maintains its keys in ascending order.

Example:

java

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

public class MapExample {
    public static void main(String[] args) {
```

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

        // Adding key-value pairs
        ages.put("Alice", 30);
        ages.put("Bob", 25);
        ages.put("Charlie", 35);

        // Accessing values by key
        System.out.println("Alice's age: " + ages.get("Alice"));

        // Iterating over the map
        for (Map.Entry<String, Integer> entry : ages.entrySet())
{
            System.out.println(entry.getKey() + " is " +
entry.getValue() + " years old.");
        }
    }
}
```

4. Queue

Purpose:

- **A `Queue` is a collection used to hold multiple elements prior to processing. It is based on the first-in-first-out (FIFO) principle, but some implementations allow elements to be processed in other orders.**

Key Implementations:

- **`LinkedList`: Implements both the `List` and `Queue` interfaces, allowing it to be used as a queue.**
- **`PriorityQueue`: A queue where elements are processed based on their priority rather than just their order in the queue.**

Example:

java

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

public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();

        // Adding elements
        queue.add("Alice");
        queue.add("Bob");
        queue.add("Charlie");

        // Processing elements in FIFO order
        System.out.println("Processing: " + queue.poll()); //
Removes and returns the head of the queue
        System.out.println("Next in line: " + queue.peek()); //
Returns the head without removing it
    }
}
```

Summary

- **List: Ordered collection, allows duplicates.**
- **Set: Unordered collection, no duplicates allowed.**
- **Map: Collection of key-value pairs, keys must be unique.**
- **Queue: Ordered collection for processing elements in a FIFO manner.**