## Java Collections Framework (JCF)

# 1. Introduction to Collections

- The **Java Collections Framework (JCF)** is a unified architecture for storing and manipulating groups of objects.
- It includes:
  - **Interfaces** (List, Set, Queue, Map)
  - **Implementations** (ArrayList, HashSet, LinkedList, HashMap, etc.)
  - **Algorithms** (Sorting, Searching, etc.)

### □ Advantages:

- Reduces programming effort.
- Increases performance.
- Provides standard interfaces.
- Supports polymorphic behavior and reduces effort to learn APIs.

---

# 2. Core Interfaces in Collections

| Interface | Description |
|---|---|
| **Collection** | Root interface of JCF |
| **List** | Ordered collection, allows duplicates |
| **Set** | Unordered, no duplicates |
| **Queue** | FIFO structure |
| **Map** | Key-value pairs |

---

# 3. List Interface

### ✔ Implementations: `ArrayList`, `LinkedList`, `Vector`, `Stack`

### Syntax:

```
List<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
list.add("Java"); // allows duplicates
```

### □ Example:

```
import java.util.*;

public class ListExample {
    public static void main(String[] args) {
        List<String> languages = new ArrayList<>();
        languages.add("Java");
        languages.add("Python");
        languages.add("JavaScript");

        for (String lang : languages) {
            System.out.println(lang);
        }
    }
}
```

---

# 4. Set Interface

### ✔ Implementations: `HashSet`, `LinkedHashSet`, `TreeSet`

- **HashSet**: No duplicates, unordered
- **LinkedHashSet**: No duplicates, maintains insertion order
- **TreeSet**: Sorted, no duplicates

### Syntax:

```
Set<String> set = new HashSet<>();
set.add("Apple");
set.add("Banana");
set.add("Apple"); // Ignored
```

### □ Example:

```
import java.util.*;

public class SetExample {
    public static void main(String[] args) {
        Set<String> fruits = new HashSet<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Mango");

        for (String fruit : fruits) {
            System.out.println(fruit);
        }
    }
}
```

---

# 5. Queue Interface

### ✔ Implementations: `LinkedList`, `PriorityQueue`

- Used in scenarios like message processing, job scheduling, etc.

### Syntax:

```
Queue<Integer> queue = new LinkedList<>();
queue.add(10);
queue.add(20);
queue.remove(); // removes head (10)
```

### □ Example:

```
import java.util.*;

public class QueueExample {
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();
        queue.add(1);
        queue.add(2);
        queue.add(3);

        System.out.println("Head: " + queue.peek());
        System.out.println("Removed: " + queue.poll());
        System.out.println("New Head: " + queue.peek());
    }
}
```

---

# 6. Map Interface

### ✔ Implementations: `HashMap`, `TreeMap`, `LinkedHashMap`, `Hashtable`

- Stores key-value pairs.

### Syntax:

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "One");
map.put(2, "Two");
```

### □ Example:

```
import java.util.*;

public class MapExample {
    public static void main(String[] args) {
        Map<Integer, String> students = new HashMap<>();
        students.put(1, "Alice");
        students.put(2, "Bob");

        for (Map.Entry<Integer, String> entry : students.entrySet()) {
            System.out.println(entry.getKey() + " => " + entry.getValue());
        }
    }
}
```

---

# 7. Iterator and ListIterator

### ✔ `Iterator` - Traverse collections (forward only)

### ✔ `ListIterator` - Forward and backward traversal (List only)

### □ Example with Iterator:

```
import java.util.*;

public class IteratorExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("A", "B", "C");
        Iterator<String> it = names.iterator();

        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

---

# 8. Sorting with Collections

### □ Example:

```
import java.util.*;

public class SortExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(4, 1, 3, 2);
        Collections.sort(numbers); // Ascending
        System.out.println("Ascending: " + numbers);

        Collections.sort(numbers, Collections.reverseOrder());
        System.out.println("Descending: " + numbers);
    }
}
```

---

# 9. Comparable vs Comparator

### ✔ Comparable (natural ordering) — uses `compareTo()`

### ✔ Comparator (custom ordering) — uses `compare()`

### □ Example: Custom sort with Comparator

```
import java.util.*;

class Student {
    String name;
    int marks;
    Student(String n, int m) { name = n; marks = m; }
}

public class ComparatorExample {
    public static void main(String[] args) {
        List<Student> list = new ArrayList<>();
        list.add(new Student("Alice", 85));
        list.add(new Student("Bob", 78));
        list.add(new Student("Charlie", 90));

        list.sort((a, b) -> b.marks - a.marks); // Descending by marks

        for (Student s : list)
            System.out.println(s.name + " => " + s.marks);
    }
}
```

---

# 10. Collections Utility Class

### ✔ Methods:

- `sort()`
- `reverse()`
- `shuffle()`
- `min(),max()`
- `frequency()`

### □ Example:

```
import java.util.*;

public class CollectionsUtilityExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("John", "Alice", "Zara");

        Collections.sort(names);
        System.out.println("Sorted: " + names);

        Collections.reverse(names);
        System.out.println("Reversed: " + names);
    }
}
```

---

# 11. Conversion between Arrays and Collections

### □ Array to List:

```
String[] arr = {"Java", "Python"};
List<String> list = Arrays.asList(arr);
```

### □ List to Array:

```
List<String> list = new ArrayList<>();
list.add("C++");
String[] arr = list.toArray(new String[0]);
```

---

# 12. Thread-safe Collections

### ✔ Legacy: `Vector`, `Hashtable`

### ✔ Wrappers:

```
List<String> syncList = Collections.synchronizedList(new ArrayList<>());
```

### ✔ Concurrent Collections:

- `ConcurrentHashMap`
- `CopyOnWriteArrayList`
- `BlockingQueue`

---

# 13. Stream API with Collections (Java 8+)

### □ Example:

```
import java.util.*;
import java.util.stream.*;

public class StreamExample {
    public static void main(String[] args) {
        List<Integer> nums = Arrays.asList(2, 4, 6, 8);
        List<Integer> squares = nums.stream()
                                    .map(n -> n * n)
                                    .collect(Collectors.toList());
        System.out.println(squares);
    }
}
```

---

# 14. Summary

| Interface | Ordered | Duplicates | Null Allowed | Example |
|---|---|---|---|---|
| List | Yes | Yes | Yes | ArrayList |
| Set | No | No | Yes (except TreeSet) | HashSet |
| Map | Keys: NoValues: Yes | No (keys) | Keys: 1 (HashMap), No (TreeMap) | HashMap |
| Queue | Yes | Yes | Yes | LinkedList |