Set Interface:

- `Set` is a **collection of unique elements** (no duplicates).
- Unlike `List`, it is **not index-based**.
- Maintains elements either unordered (`HashSet`), insertion-ordered (`LinkedHashSet`), or sorted (`TreeSet`).

(I) HashSet

- Based on **HashMap** internally.
- Stores elements in **unordered** fashion (no guarantee of order).
- Allows **null** value (only one).
- Best for **search operations** (contains, remove) since hashing makes them fast (O(1) average).

Important Methods

- `add(E e)` → Insert element (returns `false` if already exists).
- `remove(Object o)` → Remove element.
- `contains(Object o)` → Check presence.
- `size()` → Count of elements.
- `isEmpty()` → Check if empty.
- `clear()` → Remove all elements.
- `iterator()` → Traverse elements.

Example

```
import java.util.HashSet;

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

```
        set.add("Apple"); // duplicate ignored

        System.out.println("HashSet: " + set);
        System.out.println("Contains Mango? " +
set.contains("Mango"));
        set.remove("Banana");
        System.out.println("After removal: " + set);
    }
}
```

(II) LinkedHashSet

- Inherits from `HashSet` but maintains **insertion order**.
- Internally uses **LinkedHashMap**.
- Useful when order matters **and** uniqueness is required.

Important Methods (same as `HashSet`)

- `add(E e)`, `remove(Object o)`, `contains(Object o)`
- `size()`, `isEmpty()`, `clear()`
- Preserves **insertion order** unlike `HashSet`.

Example

```
import java.util.LinkedHashSet;

public class LinkedHashSetDemo {
    public static void main(String[] args) {
        LinkedHashSet<Integer> lhs = new LinkedHashSet<>();
        lhs.add(10);
        lhs.add(20);
        lhs.add(30);
        lhs.add(20); // duplicate ignored

        System.out.println("LinkedHashSet: " + lhs);
    }
```

```
}
```

(III) TreeSet

- Implements `NavigableSet` interface.
- Stores elements in **sorted (ascending) order**.
- Based on **TreeMap (Red-Black Tree)** internally.
- Doesn't allow `null` (throws `NullPointerException`).
- Useful when you need **sorted unique elements**.

Important Methods

- `add(E e)` → Insert element (keeps sorted).
- `remove(Object o)` → Remove element.
- `first()` / `last()` → Get smallest/largest element.
- `higher(E e)` → Next higher element.
- `lower(E e)` → Next lower element.
- `ceiling(E e)` → >= given element.
- `floor(E e)` → <= given element.
- `subSet(E from, E to)` → Range view.
- `headSet(E toElement)` / `tailSet(E fromElement)` → Partial views.

Example

```
import java.util.TreeSet;

public class TreeSetDemo {
    public static void main(String[] args) {
        TreeSet<Integer> ts = new TreeSet<>();
        ts.add(50);
        ts.add(20);
        ts.add(40);
        ts.add(10);

        System.out.println("TreeSet: " + ts); // Sorted
        System.out.println("First: " + ts.first());
```

```
        System.out.println("Last: " + ts.last());
        System.out.println("Higher than 20: " + ts.higher(20));
        System.out.println("SubSet(10,40): " + ts.subSet(10, 40));
    }
}
```