Priority Queue in Java

A **Priority Queue** is a special type of queue in the Java Collections Framework where elements are processed based on their priority rather than their insertion order (FIFO). Elements with higher priority are dequeued before elements with lower priority.

Why Min Heap by Default?

Priority Queue in Java is implemented using a **min heap** data structure by default. In a min heap:

- The smallest element always sits at the root (top) of the heap
- Parent nodes are always smaller than their children
- This allows O(log n) insertion and O(1) access to the minimum element

Why min heap? Because the default natural ordering in Java treats smaller values as higher priority. This is intuitive for many use cases like task scheduling (earliest deadline first) or finding minimum elements.

Key Operations

Creating a Priority Queue

java

```
PriorityQueue<Integer> pq = new PriorityQueue<>();

pq.add(5);

pq.add(2);

pq.add(8);

pq.add(1);
```

peek() - View the Top Element

java

```
System.out.println(pq.peek()); // Output: 1 (smallest element)
```

The peek() method returns the element at the root of the heap without removing it.

Printing the Priority Queue

When you print the entire Priority Queue:

java

```
System.out.println(pq); // Might output: [1, 2, 8, 5]
```

Important: The printed order doesn't show a fully sorted list! It shows the internal heap structure where:

- The first element is always the minimum
- The rest follow heap property but aren't fully sorted

poll() - Remove and Return Top Element

java

```
pq.poll(); // Removes and returns 1

System.out.println(pq); // Might output: [2, 5, 8]
```

After polling, the heap restructures itself (heapify), and the next smallest element becomes the new root. That's why when you print again, you see the new minimum element at the front.

Converting to Max Heap

To create a **max heap** (largest element has highest priority), pass Comparator.reverseOrder() during object creation:

java

```
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Comparator.reverseOrder());

maxHeap.add(5);

maxHeap.add(2);

maxHeap.add(8);

maxHeap.add(1);


System.out.println(maxHeap.peek()); // Output: 8 (largest element)
```

Now the largest element will always be at the top, and poll() will remove the maximum element first.

Example Demonstrating the Behavior

java

```
PriorityQueue<Integer> pq = new PriorityQueue<>();

pq.add(10);

pq.add(5);
pq.add(15);

pq.add(3);


System.out.println("Initial PQ: " + pq);       // [3, 5, 15, 10]

System.out.println("Peek: " + pq.peek());      // 3


pq.poll(); // Removes 3

System.out.println("After poll: " + pq);       // [5, 10, 15]

System.out.println("New peek: " + pq.peek());  // 5
```

The heap restructures after each poll(), ensuring the minimum element is always accessible in constant time while maintaining the heap property throughout the structure.

ArrayDeque

ArrayDeque in Java

ArrayDeque (Array Double Ended Queue) is a resizable array implementation of the Deque interface in Java. It allows insertion and removal of elements from both ends efficiently, making it versatile for use as a **stack**, **queue**, or **deque**.

Key Characteristics

- No capacity restrictions (grows dynamically)
- Faster than LinkedList for queue/deque operations
- Does not allow null elements
- Not thread-safe

Methods Explained

1. offer(E e) - Add to Tail

Inserts the element at the **end** of the deque (tail).

java

```
ArrayDeque<Integer> deque = new ArrayDeque<>();
```

```
deque.offer(10);

deque.offer(20);

deque.offer(30);

System.out.println(deque); // [10, 20, 30]
```

2. offerFirst(E e) - Add to Head

Inserts the element at the **beginning** of the deque (head).

java

```
deque.offerFirst(5);

System.out.println(deque); // [5, 10, 20, 30]
```

3. offerLast(E e) - Add to Tail

Inserts the element at the **end** of the deque (same as offer()).

java

```
deque.offerLast(40);

System.out.println(deque); // [5, 10, 20, 30, 40]
```

Retrieval Methods (View without Removing)

4. peek() - View First Element

Returns the **first element** (head) without removing it. Returns null if deque is empty.

java

```
System.out.println(deque.peek()); // 5

System.out.println(deque); // [5, 10, 20, 30, 40] (unchanged)
```

5. peekFirst() - View First Element

Returns the **first element** (same as peek()).

java

```
System.out.println(deque.peekFirst()); // 5
```

6. peekLast() - View Last Element

Returns the **last element** (tail) without removing it.

java

```
System.out.println(deque.peekLast()); // 40
```

Removal Methods (Remove and Return)

7. poll() - Remove First Element

Removes and returns the **first element** (head). Returns null if deque is empty.

java

```
System.out.println(deque.poll()); // 5

System.out.println(deque); // [10, 20, 30, 40]
```

8. pollFirst() - Remove First Element

Removes and returns the **first element** (same as poll()).

java

```
System.out.println(deque.pollFirst()); // 10

System.out.println(deque); // [20, 30, 40]
```

9. pollLast() - Remove Last Element

Removes and returns the **last element** (tail).

java

```
System.out.println(deque.pollLast()); // 40

System.out.println(deque); // [20, 30]
```

Complete Example

java

```
import java.util.ArrayDeque;


public class ArrayDequeDemo {

    public static void main(String[] args) {

        ArrayDeque<String> deque = new ArrayDeque<>();


        // Adding elements
        deque.offer("Middle");
        deque.offerFirst("First");
        deque.offerLast("Last");

        System.out.println("Deque: " + deque);
        // Output: [First, Middle, Last]

        // Peeking (viewing without removing)
        System.out.println("peek(): " + deque.peek());        // First
        System.out.println("peekFirst(): " + deque.peekFirst()); // First
        System.out.println("peekLast(): " + deque.peekLast());   // Last

        // Polling (removing and returning)
        System.out.println("poll(): " + deque.poll());         // First
        System.out.println("Deque: " + deque);               // [Middle, Last]

        System.out.println("pollLast(): " + deque.pollLast());   // Last
        System.out.println("Deque: " + deque);               // [Middle]

        System.out.println("pollFirst(): " + deque.pollFirst()); // Middle
        System.out.println("Deque: " + deque);               // []

        // On empty deque
        System.out.println("poll() on empty: " + deque.poll()); // null
    }
}
```

Quick Reference Table

Use Cases

- **Stack**: Use offerFirst() and pollFirst() (or push() and pop())
- **Queue**: Use offerLast() and pollFirst() (or offer() and poll())
- **Deque**: Use any combination for flexible operations

Set Interface

HashSet in Java

HashSet is a collection class in Java that implements the Set interface. It stores unique elements using a hash table (internally uses HashMap). The key feature is that it **does not allow duplicate elements**.

Key Characteristics

- **No duplicates** - Only unique elements are stored
- **Unordered** - No guarantee of element order (random arrangement)
- **Allows one null** - Can store at most one null value
- **Fast operations** - O(1) average time complexity for add, remove, contains
- **Not thread-safe** - Use Collections.synchronizedSet() for thread safety

---

Core Methods

1. add(E e) - Add Element

Adds an element to the set. Returns true if element was added, false if it already exists.

java

```
HashSet<Integer> set = new HashSet<>();
System.out.println(set.add(10)); // true (added)
System.out.println(set.add(20)); // true (added)
System.out.println(set.add(10)); // false (duplicate, not added)

System.out.println(set); // [20, 10] or [10, 20] - random order!
```

Important: The order when printed is **random/unpredictable** because HashSet doesn't maintain insertion order.

2. Cannot Add Duplicates

HashSet automatically prevents duplicate elements:

java

```
HashSet<String> names = new HashSet<>();
names.add("Alice");
names.add("Bob");
names.add("Alice"); // Won't be added again

System.out.println(names); // [Bob, Alice] or [Alice, Bob]
System.out.println(names.size()); // 2 (not 3)
```

3. Random Element Arrangement

Elements appear in unpredictable order:

java

```
HashSet<Integer> numbers = new HashSet<>();
numbers.add(5);
numbers.add(1);
numbers.add(9);
numbers.add(3);

System.out.println(numbers); // Could be [1, 3, 5, 9] or [9, 5, 1, 3] etc.
```

The order depends on hash codes and internal bucket structure, **not** insertion order.

---

4. remove(Object o) - Remove Element

Removes the specified element from the set. Returns true if element was present and removed.

java

```
HashSet<String> fruits = new HashSet<>();
```

```
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Orange");

System.out.println(fruits.remove("Banana")); // true
System.out.println(fruits.remove("Grapes")); // false (not present)

System.out.println(fruits); // [Apple, Orange] or [Orange, Apple]
```

5. contains(Object o) - Check if Element Exists

Returns true if the set contains the specified element.

java

```
HashSet<Integer> set = new HashSet<>();
set.add(100);
set.add(200);

System.out.println(set.contains(100)); // true
System.out.println(set.contains(300)); // false
```

Very efficient - O(1) average time complexity!

6. isEmpty() - Check if Set is Empty

Returns true if the set has no elements.

java

```
HashSet<String> emptySet = new HashSet<>();
System.out.println(emptySet.isEmpty()); // true

emptySet.add("Element");
System.out.println(emptySet.isEmpty()); // false
```

7. size() - Get Number of Elements

Returns the number of elements in the set.

java

```java
HashSet<Character> chars = new HashSet<>();

chars.add('A');

chars.add('B');

chars.add('C');

chars.add('A'); // Duplicate - won't be added


System.out.println(chars.size()); // 3 (not 4)
```

8. clear() - Remove All Elements

Removes all elements from the set, making it empty.

java

```java
HashSet<Double> numbers = new HashSet<>();

numbers.add(1.5);

numbers.add(2.5);

numbers.add(3.5);


System.out.println("Before clear: " + numbers); // [1.5, 2.5, 3.5]

System.out.println("Size: " + numbers.size()); // 3


numbers.clear();


System.out.println("After clear: " + numbers); // []

System.out.println("Size: " + numbers.size()); // 0

System.out.println("Is empty: " + numbers.isEmpty()); // true
```

Complete Example

java

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

```java
public class HashSetDemo {
  public static void main(String[] args) {
    HashSet<String> cities = new HashSet<>();

    // Adding elements
    System.out.println(cities.add("New York"));   // true
    System.out.println(cities.add("London"));     // true
    System.out.println(cities.add("Tokyo"));      // true
    System.out.println(cities.add("New York"));   // false (duplicate)

    // Random arrangement when printed
    System.out.println("Cities: " + cities);
    // Could be: [Tokyo, New York, London] or any other order

    // Size
    System.out.println("Size: " + cities.size()); // 3

    // Contains
    System.out.println("Contains Tokyo? " + cities.contains("Tokyo"));    // true
    System.out.println("Contains Paris? " + cities.contains("Paris"));    // false

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

    // isEmpty
    System.out.println("Is empty? " + cities.isEmpty()); // false
```

```
        // Clear
        cities.clear();
        System.out.println("After clear: " + cities);       // []
        System.out.println("Is empty? " + cities.isEmpty()); // true
        System.out.println("Size: " + cities.size());       // 0
    }
}
```

LinkedHashSet in Java

LinkedHashSet is a collection class that extends HashSet and implements the Set interface. It maintains a **doubly-linked list** of entries, which preserves the **insertion order** of elements. This is the key difference from HashSet.

Key Characteristics

- **No duplicates** - Only unique elements (like HashSet)
- **Insertion order preserved** - Elements maintain the order in which they were added
- **Allows one null** - Can store at most one null value
- **Slightly slower than HashSet** - Due to maintaining linked list, but still O(1) for most operations
- **Not thread-safe**

Core Methods (Same as HashSet)

1. add(E e) - Add Element

Adds an element to the set while **preserving insertion order**. Returns true if element was added, false if duplicate.

java

```
LinkedHashSet<Integer> set = new LinkedHashSet<>();
System.out.println(set.add(10)); // true
System.out.println(set.add(20)); // true
System.out.println(set.add(30)); // true
System.out.println(set.add(10)); // false (duplicate)

System.out.println(set); // [10, 20, 30] - ORDER PRESERVED!
```

Key Difference from HashSet: Elements print in the **exact order** they were inserted, not random!

2. Order of Elements Reserved (Preserved)

The biggest advantage - insertion order is maintained:

java

```
LinkedHashSet<String> names = new LinkedHashSet<>();
names.add("Alice");
names.add("Bob");
names.add("Charlie");
names.add("Diana");

System.out.println(names); // [Alice, Bob, Charlie, Diana]
// Always prints in this order!
```

Compare with HashSet:

java

```
HashSet<String> hashSet = new HashSet<>();
hashSet.add("Alice");
hashSet.add("Bob");
hashSet.add("Charlie");

System.out.println(hashSet); // [Bob, Charlie, Alice] - RANDOM ORDER!
```

3. Cannot Add Duplicates

Just like HashSet, duplicates are not allowed:

java

```
LinkedHashSet<Integer> numbers = new LinkedHashSet<>();

numbers.add(5);

numbers.add(10);

numbers.add(5);  // Duplicate - won't be added

numbers.add(15);


System.out.println(numbers); // [5, 10, 15] - in insertion order

System.out.println(numbers.size()); // 3
```

Important: If you try to add a duplicate, it doesn't change the position of the existing element.

---

4. remove(Object o) - Remove Element

Removes the specified element. Returns true if removed. **Order of remaining elements stays the same.**

java

```
LinkedHashSet<String> fruits = new LinkedHashSet<>();

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Orange");

fruits.add("Mango");


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

// [Apple, Banana, Orange, Mango]


fruits.remove("Banana");

System.out.println("After removing Banana: " + fruits);
```

```java
// [Apple, Orange, Mango] - order maintained!

System.out.println(fruits.remove("Grapes")); // false
```

5. contains(Object o) - Check if Element Exists

Returns true if the set contains the specified element.

java

```java
LinkedHashSet<Integer> set = new LinkedHashSet<>();
set.add(100);
set.add(200);
set.add(300);

System.out.println(set.contains(200)); // true
System.out.println(set.contains(400)); // false

System.out.println(set); // [100, 200, 300] - order maintained
```

6. isEmpty() - Check if Set is Empty

Returns true if the set has no elements.

java

```java
LinkedHashSet<String> emptySet = new LinkedHashSet<>();
System.out.println(emptySet.isEmpty()); // true

emptySet.add("First");
System.out.println(emptySet.isEmpty()); // false
```

7. size() - Get Number of Elements

Returns the number of elements in the set.

java

```java
LinkedHashSet<Character> chars = new LinkedHashSet<>();
chars.add('A');
chars.add('B');
chars.add('C');
chars.add('A'); // Duplicate - won't increase size

System.out.println(chars.size()); // 3
System.out.println(chars); // [A, B, C]
```

8. clear() - Remove All Elements

Removes all elements from the set.

java

```
LinkedHashSet<Double> numbers = new LinkedHashSet<>();
numbers.add(1.1);
numbers.add(2.2);
numbers.add(3.3);

System.out.println("Before: " + numbers); // [1.1, 2.2, 3.3]
System.out.println("Size: " + numbers.size()); // 3

numbers.clear();

System.out.println("After: " + numbers); // []
System.out.println("Size: " + numbers.size()); // 0
System.out.println("Is empty: " + numbers.isEmpty()); // true
```

Complete Example Demonstrating Order Preservation

java

```
import java.util.LinkedHashSet;
import java.util.HashSet;
```

```
public class LinkedHashSetDemo {
  public static void main(String[] args) {
    // LinkedHashSet - Order Preserved
    System.out.println("=== LinkedHashSet ===");
    LinkedHashSet<String> linkedSet = new LinkedHashSet<>();

    linkedSet.add("First");
    linkedSet.add("Second");
    linkedSet.add("Third");
    linkedSet.add("Fourth");
    linkedSet.add("Second"); // Duplicate - won't be added

    System.out.println("LinkedHashSet: " + linkedSet);
    // Output: [First, Second, Third, Fourth] - ALWAYS THIS ORDER

    // Remove element
    linkedSet.remove("Second");
    System.out.println("After removing Second: " + linkedSet);
    // [First, Third, Fourth] - order maintained

    // Other operations
    System.out.println("Contains 'Third'? " + linkedSet.contains("Third")); // true
    System.out.println("Size: " + linkedSet.size()); // 3
    System.out.println("Is empty? " + linkedSet.isEmpty()); // false

    linkedSet.clear();
    System.out.println("After clear: " + linkedSet); // []
    // Compare with HashSet - Random Order
    System.out.println("\n=== HashSet (for comparison) ===");
    HashSet<String> hashSet = new HashSet<>();

    hashSet.add("First");
    hashSet.add("Second");
    hashSet.add("Third");
    hashSet.add("Fourth");

    System.out.println("HashSet: " + hashSet);
    // Output: Random order like [Third, First, Fourth, Second]
  }
}
```

Practical Example: Removing Duplicates While Preserving Order

java

```
import java.util.LinkedHashSet;
import java.util.Arrays;
import java.util.List;

public class RemoveDuplicates {
  public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(5, 2, 8, 2, 9, 5, 3, 8, 1);

    System.out.println("Original: " + numbers);
    // [5, 2, 8, 2, 9, 5, 3, 8, 1]

    LinkedHashSet<Integer> uniqueNumbers = new LinkedHashSet<>(numbers);
```

```
        System.out.println("After removing duplicates: " + uniqueNumbers);
        // [5, 2, 8, 9, 3, 1] - duplicates removed, order preserved!
    }
}
```

TreeSet in Java

TreeSet is a collection class that implements the NavigableSet interface (which extends SortedSet). It stores elements in a **sorted (ascending) order** using a **Red-Black tree** data structure. Elements are automatically sorted based on their natural ordering or a custom comparator.

Key Characteristics

- **No duplicates** - Only unique elements (like HashSet)
- **Sorted order** - Elements automatically sorted in ascending order
- **No null allowed** - Cannot store null values (throws NullPointerException)
- **Slower than HashSet/LinkedHashSet** - O(log n) time for add, remove, contains
- **Not thread-safe**
- Elements must be **comparable** (implement Comparable) or use a custom Comparator

---

Core Methods

1. add(E e) - Add Element

Adds an element to the set in **sorted order**. Returns true if element was added, false if duplicate.

java

```
TreeSet<Integer> set = new TreeSet<>();
System.out.println(set.add(30)); // true
System.out.println(set.add(10)); // true
System.out.println(set.add(20)); // true
System.out.println(set.add(10)); // false (duplicate)

System.out.println(set); // [10, 20, 30] - SORTED ORDER!
```

Key Point: No matter what order you add elements, TreeSet automatically sorts them!

2. Elements Are Always Sorted

The most important feature - elements are always in ascending order:

java

```
TreeSet<String> names = new TreeSet<>();
names.add("Diana");
names.add("Alice");
names.add("Charlie");
names.add("Bob");

System.out.println(names); // [Alice, Bob, Charlie, Diana] - SORTED!
```

Comparison with other Sets:

java

```
// HashSet - Random order
HashSet<Integer> hashSet = new HashSet<>();
hashSet.add(50); hashSet.add(10); hashSet.add(30);
System.out.println(hashSet); // [50, 10, 30] or random

// LinkedHashSet - Insertion order
LinkedHashSet<Integer> linkedSet = new LinkedHashSet<>();
linkedSet.add(50); linkedSet.add(10); linkedSet.add(30);
System.out.println(linkedSet); // [50, 10, 30] - as inserted
```

```
// TreeSet - Sorted order
TreeSet<Integer> treeSet = new TreeSet<>();
treeSet.add(50); treeSet.add(10); treeSet.add(30);
System.out.println(treeSet); // [10, 30, 50] - SORTED!
```

3. Cannot Add Duplicates

Duplicates are not allowed (same as other Sets):

java

```
TreeSet<Integer> numbers = new TreeSet<>();
numbers.add(15);
numbers.add(5);
numbers.add(25);
numbers.add(5);  // Duplicate - won't be added
numbers.add(10);

System.out.println(numbers); // [5, 10, 15, 25] - sorted, no duplicates
System.out.println(numbers.size()); // 4
```

4. remove(Object o) - Remove Element

Removes the specified element. Returns true if removed. **Remaining elements stay sorted.**

java

```
TreeSet<String> fruits = new TreeSet<>();
fruits.add("Orange");
fruits.add("Apple");
fruits.add("Mango");
fruits.add("Banana");

System.out.println("Before: " + fruits);
// [Apple, Banana, Mango, Orange] - sorted
```

```
fruits.remove("Banana");
System.out.println("After removing Banana: " + fruits);
// [Apple, Mango, Orange] - still sorted!

System.out.println(fruits.remove("Grapes")); // false
```

5. contains(Object o) - Check if Element Exists

Returns true if the set contains the specified element. Uses binary search internally (faster than linear search).

java

```
TreeSet<Integer> set = new TreeSet<>();
set.add(100);
set.add(200);
set.add(300);
set.add(150);

System.out.println(set); // [100, 150, 200, 300]
System.out.println(set.contains(200)); // true
System.out.println(set.contains(250)); // false
```

6. isEmpty() - Check if Set is Empty

Returns true if the set has no elements.

java

```
TreeSet<String> emptySet = new TreeSet<>();
System.out.println(emptySet.isEmpty()); // true

emptySet.add("Element");
System.out.println(emptySet.isEmpty()); // false
```

7. size() - Get Number of Elements

Returns the number of elements in the set.

java

```
TreeSet<Character> chars = new TreeSet<>();
chars.add('C');
chars.add('A');
chars.add('B');
chars.add('A'); // Duplicate - won't increase size

System.out.println(chars.size()); // 3
System.out.println(chars); // [A, B, C] - sorted
```

8. clear() - Remove All Elements

Removes all elements from the set.

java

```
TreeSet<Double> numbers = new TreeSet<>();
numbers.add(3.3);
numbers.add(1.1);
numbers.add(2.2);

System.out.println("Before: " + numbers); // [1.1, 2.2, 3.3]
System.out.println("Size: " + numbers.size()); // 3

numbers.clear();

System.out.println("After: " + numbers); // []
System.out.println("Size: " + numbers.size()); // 0
System.out.println("Is empty: " + numbers.isEmpty()); // true
```

Additional TreeSet-Specific Methods

TreeSet has extra methods for navigating sorted elements:

first() - Get First (Smallest) Element

java

```
TreeSet<Integer> set = new TreeSet<>();

set.add(50); set.add(20); set.add(80);

System.out.println(set.first()); // 20
```

last() - Get Last (Largest) Element

java

```
System.out.println(set.last()); // 80
```

pollFirst() - Remove and Return First Element

java

```
System.out.println(set.pollFirst()); // 20

System.out.println(set); // [50, 80]
```

pollLast() - Remove and Return Last Element

java

```
System.out.println(set.pollLast()); // 80

System.out.println(set); // [50]
```

higher(E e) - Get Next Higher Element

java

```
TreeSet<Integer> nums = new TreeSet<>();

nums.add(10); nums.add(20); nums.add(30); nums.add(40);

System.out.println(nums.higher(20)); // 30

System.out.println(nums.higher(25)); // 30
```

lower(E e) - Get Next Lower Element

java

```
System.out.println(nums.lower(30)); // 20

System.out.println(nums.lower(25)); // 20
```

Complete Example

java

```
import java.util.TreeSet;

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

        // Adding elements - they get sorted automatically
        numbers.add(50);
        numbers.add(20);
        numbers.add(80);
        numbers.add(10);
        numbers.add(40);
        numbers.add(20); // Duplicate - won't be added

        System.out.println("TreeSet: " + numbers);
        // Output: [10, 20, 40, 50, 80] - ALWAYS SORTED!

        // Size
        System.out.println("Size: " + numbers.size()); // 5

        // Contains
        System.out.println("Contains 40? " + numbers.contains(40)); // true
        System.out.println("Contains 30? " + numbers.contains(30)); // false

        // Remove
        numbers.remove(40);
        System.out.println("After removing 40: " + numbers);
        // [10, 20, 50, 80] - still sorted

        // isEmpty
        System.out.println("Is empty? " + numbers.isEmpty()); // false

        // TreeSet specific methods
        System.out.println("First element: " + numbers.first()); // 10
        System.out.println("Last element: " + numbers.last());   // 80
        System.out.println("Higher than 20: " + numbers.higher(20)); // 50
        System.out.println("Lower than 50: " + numbers.lower(50));   // 20

        // Clear
        numbers.clear();
        System.out.println("After clear: " + numbers); // []
        System.out.println("Is empty? " + numbers.isEmpty()); // true
    }
}
```

Using Custom Comparator (Reverse Order)

java

```
import java.util.TreeSet;
import java.util.Comparator;

public class TreeSetReverseOrder {
    public static void main(String[] args) {
        // TreeSet in descending order
        TreeSet<Integer> descSet = new TreeSet<>(Comparator.reverseOrder());
        descSet.add(50);
        descSet.add(20);
        descSet.add(80);
        descSet.add(10);

        System.out.println(descSet); // [80, 50, 20, 10] - REVERSE SORTED!
    }
}
```

Custom Objects in TreeSet

For custom objects, implement Comparable or provide a Comparator:

java

```
import java.util.TreeSet;

class Student implements Comparable<Student> {
    String name;
    int marks;

    Student(String name, int marks) {
        this.name = name;
        this.marks = marks;
    }

    @Override
    public int compareTo(Student other) {
        return this.marks - other.marks; // Sort by marks
    }
    @Override
    public String toString() {
        return name + "(" + marks + ")";
    }
}

public class CustomTreeSet {
    public static void main(String[] args) {
        TreeSet<Student> students = new TreeSet<>();

        students.add(new Student("Alice", 85));
        students.add(new Student("Bob", 70));
        students.add(new Student("Charlie", 95));
        students.add(new Student("Diana", 80));

        System.out.println(students);
        // [Bob(70), Diana(80), Alice(85), Charlie(95)] - sorted by marks!
    }
}
```

HashSet with Custom Objects (Student Class)

When working with custom objects in a HashSet, understanding hashCode() and equals() is crucial for proper duplicate detection.

The Problem: Before Implementing hashCode() and equals()

Let me first explain what happens WITHOUT proper hashCode() and equals() implementation:

Student Class WITHOUT hashCode() and equals()

java

```
class Student {
    String name;
    int rollNo;

    Student(String name, int rollNo) {
        this.name = name;
        this.rollNo = rollNo;
    }

    @Override
    public String toString() {
        return "Student{name='" + name + "', rollNo=" + rollNo + "}";
    }
}
```

What Goes Wrong?

java

```
import java.util.HashSet;

public class BeforeHashCode {
    public static void main(String[] args) {
        HashSet<Student> students = new HashSet<>();

        Student s1 = new Student("Alice", 101);
        Student s2 = new Student("Bob", 102);
        Student s3 = new Student("Alice", 101); // Same as s1!

        students.add(s1);
        students.add(s2);
        students.add(s3); // Should be duplicate, but gets added!

        System.out.println("Size: " + students.size()); // 3 (WRONG! Should be 2)
        System.out.println(students);

        // Output shows 3 students even though s1 and s3 are duplicates:
        // [Student{name='Bob', rollNo=102},
        //  Student{name='Alice', rollNo=101},
        //  Student{name='Alice', rollNo=101}]
    }
}
```

Why This Happens?

Without proper hashCode() and equals():

- HashSet uses the **default hashCode()** from Object class, which is based on memory address
- Each object has a different memory address, so s1 and s3 have different hash codes
- HashSet thinks they are different objects, even though logically they represent the same student (same rollNo)
- **Result:** Duplicates are allowed! ❌

The Solution: Implementing hashCode() and equals()

Complete Student Class WITH hashCode() and equals()

java

```
import java.util.Objects;
```

```
class Student {
  String name;
  int rollNo;

  Student(String name, int rollNo) {
    this.name = name;
    this.rollNo = rollNo;
  }

  // toString() - For readable output when printing
  @Override
  public String toString() {
    return "Student{name='" + name + "', rollNo=" + rollNo + "}";
  }

  // hashCode() - Calculates hash based on rollNo
  // Since rollNo is unique, we use it to identify each student
  @Override
  public int hashCode() {
    return Objects.hash(rollNo); // Hash based on rollNo only
  }

  // equals() - Compares students based on rollNo
  // Two students are equal if they have the same rollNo
  @Override
  public boolean equals(Object obj) {
    // Check if same object reference
    if (this == obj) return true;

    // Check if null or different class
    if (obj == null || getClass() != obj.getClass()) return false;

    // Cast to Student and compare rollNo
    Student other = (Student) obj;
    return this.rollNo == other.rollNo;
  }
}
```

How It Works Now:

java

```
import java.util.HashSet;

public class AfterHashCode {
  public static void main(String[] args) {
    HashSet<Student> students = new HashSet<>();

    Student s1 = new Student("Alice", 101);
    Student s2 = new Student("Bob", 102);
    Student s3 = new Student("Alice", 101); // Same rollNo as s1
    Student s4 = new Student("Charlie", 101); // Different name, same rollNo as s1

    System.out.println(students.add(s1)); // true - added
    System.out.println(students.add(s2)); // true - added
    System.out.println(students.add(s3)); // false - DUPLICATE! Not added
    System.out.println(students.add(s4)); // false - DUPLICATE! Not added
    System.out.println("\nSize: " + students.size()); // 2 (CORRECT!)
    System.out.println(students);

    // Output shows only 2 unique students:
    // [Student{name='Bob', rollNo=102},
    //  Student{name='Alice', rollNo=101}]
  }
}
```

Now it works correctly! ✅

- s3 is recognized as duplicate of s1 (same rollNo: 101)
- s4 is also recognized as duplicate (same rollNo: 101, even with different name)
- HashSet properly prevents duplicates based on rollNo

Detailed Explanation of hashCode() Method

Why We Need hashCode()?

HashSet internally uses a **hash table** structure:

1. **When adding an element:**
   - HashSet calls hashCode() on the object
   - Uses the hash code to determine which "bucket" to place the object in
   - If bucket already has objects, it uses equals() to check for duplicates
2. **The Contract:**
   - If two objects are equal (according to equals()), they **must** have the same hash code
   - If two objects have the same hash code, they **may or may not** be equal

Our Implementation:

java

```
@Override
public int hashCode() {
    return Objects.hash(rollNo);
}
```

Explanation:

- We use Objects.hash() utility method (requires import java.util.Objects)
- We hash based on rollNo because that's what makes each student unique
- All students with the same rollNo will have the **same hash code**
- This ensures HashSet can detect duplicates

Alternative Implementations:

java

```
// Option 1: Simple hash based on rollNo
@Override
public int hashCode() {
    return rollNo;
}

// Option 2: If both name and rollNo should be considered
@Override
public int hashCode() {
    return Objects.hash(name, rollNo);
}

// Option 3: Manual calculation (old style)
@Override
public int hashCode() {
    int result = 17;
    result = 31 * result + rollNo;
    return result;
}
```

Detailed Explanation of equals() Method

Why We Need equals()?

The equals() method determines if two objects are logically equal:

java

```
@Override
public boolean equals(Object obj) {
    // Step 1: Check if same object reference (optimization)
    if (this == obj) return true;

    // Step 2: Check if null or different class type
    if (obj == null || getClass() != obj.getClass()) return false;

    // Step 3: Cast and compare the unique identifier (rollNo)
    Student other = (Student) obj;
    return this.rollNo == other.rollNo;
}
```

Step-by-step breakdown:

1. **if (this == obj)** - If both references point to the same object, they're obviously equal
2. **if (obj == null)** - Null is never equal to a real object
3. **getClass() != obj.getClass()** - Make sure we're comparing two Student objects
4. **this.rollNo == other.rollNo** - The actual comparison based on our business logic

---

Complete Working Example

java

```
import java.util.HashSet;
import java.util.Objects;
```

```
class Student {
    String name;
    int rollNo;

    Student(String name, int rollNo) {
        this.name = name;
        this.rollNo = rollNo;
    }

    @Override
    public String toString() {
        return "Student{name='" + name + "', rollNo=" + rollNo + "}";
    }

    @Override
    public int hashCode() {
        return Objects.hash(rollNo);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Student other = (Student) obj;
        return this.rollNo == other.rollNo;
    }
}
```

```
public class StudentHashSetDemo {
    public static void main(String[] args) {
        HashSet<Student> students = new HashSet<>();

        // Adding students
        students.add(new Student("Alice", 101));
        students.add(new Student("Bob", 102));
        students.add(new Student("Charlie", 103));
        students.add(new Student("Diana", 104));

        // Try adding duplicate rollNo (different name)
        students.add(new Student("Alice Smith", 101)); // Won't be added

        // Try adding duplicate rollNo (same name)
        students.add(new Student("Bob", 102)); // Won't be added

        System.out.println("Total students: " + students.size()); // 4

        // Print all students (thanks to toString())
        System.out.println("\nAll Students:");
        for (Student s : students) {
            System.out.println(s);
        }

        // Check if student exists
        Student searchStudent = new Student("Test", 102);
        System.out.println("\nContains student with rollNo 102? "
                + students.contains(searchStudent)); // true

        // Remove a student
        students.remove(new Student("AnyName", 103)); // Removes Charlie
        System.out.println("\nAfter removing rollNo 103:");
        System.out.println(students);
    }
}
```

Output:

```
Total students: 4

All Students:
Student{name='Diana', rollNo=104}
Student{name='Bob', rollNo=102}
Student{name='Alice', rollNo=101}
Student{name='Charlie', rollNo=103}

Contains student with rollNo 102? true

After removing rollNo 103:
[Student{name='Diana', rollNo=104}, Student{name='Bob', rollNo=102},
Student{name='Alice', rollNo=101}]
```

Before vs After Comparison

Scenario	**Without hashCode()/equals()**	**With hashCode()/equals()**
Add duplicate rollNo	✗ Allows duplicates	✓ Prevents duplicates
contains() method	✗ Can't find logically equal objects	✓ Finds based on rollNo
remove() method	✗ Can't remove using new object	✓ Can remove using rollNo
Memory usage	✗ Wastes memory on duplicates	✓ Efficient storage