Set Interface

In Java, a **Set** is a collection that contains no duplicate elements. The `Set` interface is part of the Java Collections Framework and has several implementations, the most common being **HashSet**, **LinkedHashSet**, and **TreeSet**.

Let's explore these implementations with simple examples for each method in the **Set** interface.

Set Interface and Methods

Here's a list of some commonly used methods in the **Set** interface:

- **add()**
- **addAll()**
- **clear()**
- **contains(Object o)**
- **isEmpty()**
- **iterator()**
- **remove(Object o)**
- **size()**

HashSet Example

A **HashSet** is backed by a hash table and does not guarantee any specific iteration order.

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

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

        Set<String> hashSet = new HashSet<>();


        // add()
        hashSet.add("Apple");
        hashSet.add("Banana");
        hashSet.add("Cherry");
        System.out.println("HashSet after add: " + hashSet);



        // contains()System.out.println("Contains
       'Banana':"+hashSet.contains("Banana"));
```

```
        // size()
        System.out.println("Size: " + hashSet.size());

        // remove()
        hashSet.remove("Banana");
        System.out.println("HashSet after remove: " + hashSet);


        // isEmpty()
        System.out.println("Is empty: " + hashSet.isEmpty());


        // iterator()
        System.out.print("Iterating: ");
        for (String item : hashSet) {
            System.out.print(item + " ");
        }
        System.out.println();


        // clear()
        hashSet.clear();
        System.out.println("HashSet after clear: " + hashSet);
    }
}
```

LinkedHashSet

A **LinkedHashSet** maintains a linked list of the entries in the set, which defines the iteration order (**insertion order**).

java
Copy code

```
import java.util.LinkedHashSet;
import java.util.Set;

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

      Set<String> linkedHashSet = new LinkedHashSet<>();

      // add()
      linkedHashSet.add("Apple");
      linkedHashSet.add("Banana");
      linkedHashSet.add("Cherry");
```

```
        System.out.println("LinkedHashSet after add: " + linkedHashSet);

        // contains()
        System.out.println("Contains 'Banana': " +
linkedHashSet.contains("Banana"));


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


        // remove()
        linkedHashSet.remove("Banana");
        System.out.println("LinkedHashSet after remove: " + linkedHashSet);


        // isEmpty()
        System.out.println("Is empty: " + linkedHashSet.isEmpty());


        // iterator()
        System.out.print("Iterating: ");
        for (String item : linkedHashSet) {
            System.out.print(item + " ");
        }
        System.out.println();


        // clear()
        linkedHashSet.clear();
        System.out.println("LinkedHashSet after clear: " + linkedHashSet);
    }
}
```

SortedSet Interface

The **SortedSet** interface in Java is a subinterface of **Set** that provides a total ordering on its elements. The **TreeSet** class is a concrete implementation of the `SortedSet` interface, which means it stores elements in a sorted order.

The **SortedSet** interface provides additional methods to handle the sorted nature of the set:

- **Comparator<? super E> comparator()**
- **first()**
- **SortedSet<E> headSet(E toElement)**
- **last()**
- **SortedSet<E> subSet(E fromElement, E toElement)**
- **SortedSet<E> tailSet(E fromElement)**

SortedSet

Here's an example that demonstrates the usage of the **SortedSet** interface with the **TreeSet** class:

java
Copy code

```
import java.util.SortedSet;
import java.util.TreeSet;

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

      SortedSet<String> sortedSet = new TreeSet<>();

      // add()
      sortedSet.add("Apple");
      sortedSet.add("Banana");
      sortedSet.add("Cherry");
      sortedSet.add("Date");
      sortedSet.add("Fig");
     System.out.println("SortedSet after add: " + sortedSet);

      // first()
      System.out.println("First element: " + sortedSet.first());


      // last()
      System.out.println("Last element: " + sortedSet.last());


      // headSet(E toElement)
      SortedSet<String> headSet = sortedSet.headSet("Cherry");
      System.out.println("HeadSet (elements before 'Cherry'): " + headSet);


      // tailSet(E fromElement)
```

```
        SortedSet<String> tailSet = sortedSet.tailSet("Cherry");
        System.out.println("TailSet (elements from 'Cherry' onwards): " + tailSet);


        // subSet(E fromElement, E toElement)
        SortedSet<String> subSet = sortedSet.subSet("Banana", "Fig");
        System.out.println("SubSet (elements from 'Banana' to 'Fig'): " + subSet);


        // comparator()
        System.out.println("Comparator: " + sortedSet.comparator()); // Should print "null"
if natural ordering is used


        // iterator()
        System.out.print("Iterating: ");
        for (String item : sortedSet) {
            System.out.print(item + " ");
        }
        System.out.println();
    }
}
```

Explanation:

1. **Creating a SortedSet**:
    - **TreeSet** is used as the concrete implementation of **SortedSet**.
2. **Adding Elements**:
    - Elements are added using the **add()** method. They are automatically sorted in natural order.
3. **Accessing First and Last Elements**:
    - **first()** retrieves the smallest element.
    - **last()** retrieves the largest element.
4. **Creating Subsets**:
    - **headSet(E toElement):** Returns a view of the portion of this set whose elements are strictly less than **toElement**.
    - **tailSet(E fromElement):** Returns a view of the portion of this set whose elements are greater than or equal to **fromElement.**
    - **subSet(E fromElement, E toElement):** Returns a view of the portion of this set whose elements range from **fromElement,** inclusive, to **toElement,** exclusive.
5. **Comparator**:
    - **comparator()** returns the comparator used to order the elements in this set, or **null** if it uses natural ordering.
6. **Iteration**:
    - The **iterator()** method is used to iterate over the elements of the set.

Summary

- **SortedSet**: Ensures that the elements are in a sorted order.
- **TreeSet**: A concrete implementation of **SortedSet** which maintains elements in a sorted order using a Red-Black tree.

This example illustrates how to use the **SortedSet** interface and its methods to manipulate and retrieve elements in a sorted manner.

TreeSet

A **TreeSet** is a NavigableSet implementation based on a TreeMap. The elements in a **TreeSet** are sorted.

java
Copy code

```
import java.util.Set;
import java.util.TreeSet;
public class TreeSetExample {
   public static void main(String[] args) {

      Set<String> treeSet = new TreeSet<>();

      // add()
      treeSet.add("Apple");
      treeSet.add("Banana");
      treeSet.add("Cherry");
      System.out.println("TreeSet after add: " + treeSet);

      // contains()
      System.out.println("Contains 'Banana': " + treeSet.contains("Banana"));

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

      // remove()
      treeSet.remove("Banana");
      System.out.println("TreeSet after remove: " + treeSet);

      // isEmpty()
```

```
        System.out.println("Is empty: " + treeSet.isEmpty());

        // iterator()
        System.out.print("Iterating: ");
        for (String item : treeSet) {
            System.out.print(item + " ");
        }
        System.out.println();

        // clear()
        treeSet.clear();
        System.out.println("TreeSet after clear: " + treeSet);
    }
}
```

Summary

- **HashSet**: Provides no guarantee on the order of elements. Suitable when you don't care about the order.
- **LinkedHashSet**: Maintains the order of insertion. Use this when you need to maintain the insertion order.
- **TreeSet**: Maintains elements in sorted order. Use this when you need to keep elements sorted.

These examples cover the basic usage of the `Set` interface and its commonly used methods. You can expand on these examples by exploring other methods and properties of the **Set** interface and its implementations.