Java Set Interface

The Set interface of the Java Collections framework provides the features of the mathematical set in Java. It extends the Collection interface.

Unlike the List interface, sets cannot contain duplicate elements.

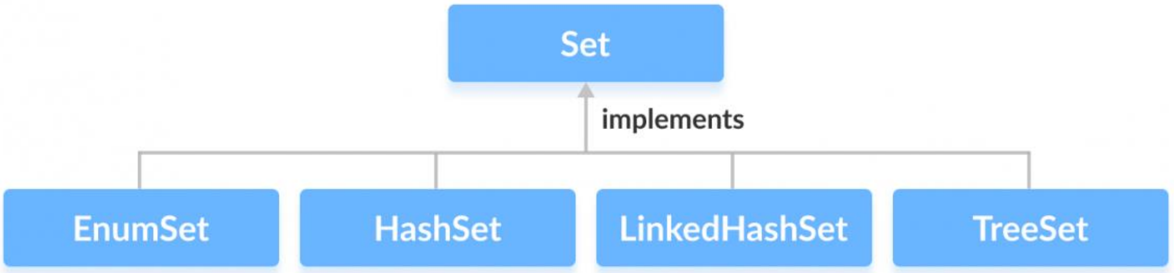
Classes that implement Set

Since Set is an interface, we cannot create objects from it.

In order to use functionalities of the Set interface, we can use these classes:

- HashSet
- LinkedHashSet
- EnumSet
- TreeSet

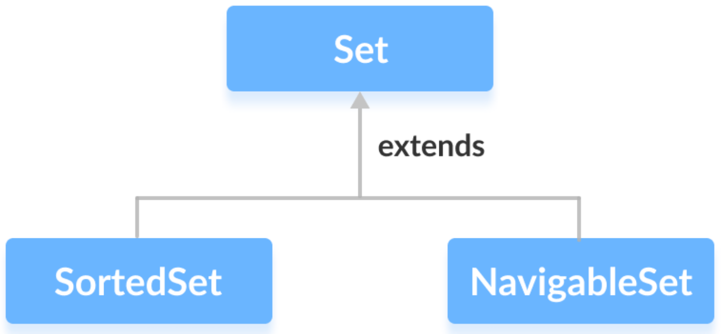
These classes are defined in the Collections framework and implement the Set interface.



Interfaces that extend Set

The Set interface is also extended by these subinterfaces:

- SortedSet
- NavigableSet



How to use Set?

In Java, we must import `java.util.Set` package in order to use `Set`.

```
// Set implementation using HashSet
Set<String> animals = new HashSet<>();
```

Here, we have created a Set called *animals*. We have used the HashSet class to implement the Set interface.

Methods of Set

The Set interface includes all the methods of the Collection interface. It's because Collection is a super interface of Set.

Some of the commonly used methods of the Collection interface that's also available in the Set interface are:

- **add()** - adds the specified element to the set
- **addAll()** - adds all the elements of the specified collection to the set
- **iterator()** - returns an iterator that can be used to access elements of the set sequentially
- **remove()** - removes the specified element from the set
- **removeAll()** - removes all the elements from the set that is present in another specified set
- **retainAll()** - retains all the elements in the set that are also present in another specified set
- **clear()** - removes all the elements from the set
- **size()** - returns the length (number of elements) of the set
- **toArray()** - returns an array containing all the elements of the set

- **contains()** - returns `true` if the set contains the specified element
- **containsAll()** - returns `true` if the set contains all the elements of the specified collection
- **hashCode()** - returns a hash code value (address of the element in the set)

Set Operations

The Java `Set` interface allows us to perform basic mathematical set operations like union, intersection, and subset.

- **Union** - to get the union of two sets *x* and *y*, we can use `x.addAll(y)`
- **Intersection** - to get the intersection of two sets *x* and *y*, we can use `x.retainAll(y)`
- **Subset** - to check if *x* is a subset of *y*, we can use `y.containsAll(x)`

Implementation of the Set Interface

1. Implementing HashSet Class

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

class Main {

    public static void main(String[] args) {
        // Creating a set using the HashSet class
        Set<Integer> set1 = new HashSet<>();

        // Add elements to the set1
        set1.add(2);
        set1.add(3);
        System.out.println("Set1: " + set1);

        // Creating another set using the HashSet class
        Set<Integer> set2 = new HashSet<>();

        // Add elements
        set2.add(1);
        set2.add(2);
        System.out.println("Set2: " + set2);

        // Union of two sets
        set2.addAll(set1);
        System.out.println("Union is: " + set2);
    }
}
```

Output

```
Set1: [2, 3]
Set2: [1, 2]
Union is: [1, 2, 3]
```

2. Implementing TreeSet Class

```
import java.util.Set;
import java.util.TreeSet;
import java.util.Iterator;

class Main {

    public static void main(String[] args) {
        // Creating a set using the TreeSet class
        Set<Integer> numbers = new TreeSet<>();

        // Add elements to the set
        numbers.add(2);
        numbers.add(3);
        numbers.add(1);
        System.out.println("Set using TreeSet: " + numbers);

        // Access Elements using iterator()
        System.out.print("Accessing elements using iterator(): ");
        Iterator<Integer> iterate = numbers.iterator();
        while(iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }

    }
}
```

Output

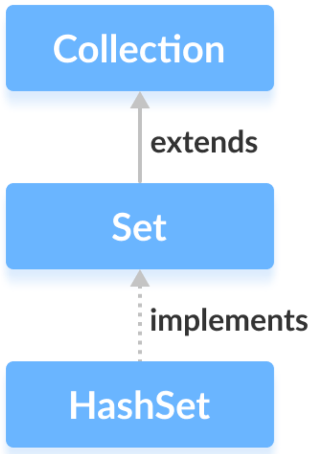
```
Set using TreeSet: [1, 2, 3]
Accessing elements using iterator(): 1, 2, 3,
```

Now that we know what Set is, we will see its implementations in classes like EnumSet, HashSet, LinkedHashSet and TreeSet in the next tutorials.

Java HashSet Class

The HashSet class of the Java Collections framework provides the functionalities of the hash table data structure.

It implements the Set interface.



Creating a HashSet

In order to create a hash set, we must import the java.util.HashSet package first.

Once we import the package, here is how we can create hash sets in Java.

```
// HashSet with 8 capacity and 0.75 load factor
HashSet<Integer> numbers = new HashSet<>(8, 0.75);
```

Here, we have created a hash set named numbers.

Notice, the part new HashSet<>(8, 0.75). Here, the first parameter is **capacity**, and the second parameter is **loadFactor**.

- **capacity** - The capacity of this hash set is 8. Meaning, it can store 8 elements.
- **loadFactor** - The load factor of this hash set is 0.6. This means, whenever our hash set is filled by 60%, the elements are moved to a new hash table of double the size of the original hash table.

Default capacity and load factor

It's possible to create a hash table without defining its capacity and load factor. For example,

```
// HashSet with default capacity and load factor
HashSet<Integer> numbers1 = new HashSet<>();
```

By default,

- the capacity of the hash set will be 16
- the load factor will be 0.75

Methods of HashSet

The `HashSet` class provides various methods that allow us to perform various operations on the set.

Insert Elements to HashSet

- **`add()`** - inserts the specified element to the set
- **`addAll()`** - inserts all the elements of the specified collection to the set

For example,

```
import java.util.HashSet;

class Main {
    public static void main(String[] args) {
        HashSet<Integer> evenNumber = new HashSet<>();

        // Using add() method
        evenNumber.add(2);
        evenNumber.add(4);
        evenNumber.add(6);
        System.out.println("HashSet: " + evenNumber);

        HashSet<Integer> numbers = new HashSet<>();

        // Using addAll() method
        numbers.addAll(evenNumber);
        numbers.add(5);
        System.out.println("New HashSet: " + numbers);
    }
}
```

Output

```
HashSet: [2, 4, 6]
New HashSet: [2, 4, 5, 6]
```

Access HashSet Elements

To access the elements of a hash set, we can use the iterator() method. In order to use this method, we must import the java.util.Iterator package. For example,

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

class Main {
    public static void main(String[] args) {
        HashSet<Integer> numbers = new HashSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(6);
        System.out.println("HashSet: " + numbers);

        // Calling iterator() method
        Iterator<Integer> iterate = numbers.iterator();
        System.out.print("HashSet using Iterator: ");
        // Accessing elements
        while(iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }
    }
}
```

Output

```
HashSet: [2, 5, 6]
HashSet using Iterator: 2, 5, 6,
```

Remove Elements

- remove() - removes the specified element from the set
- removeAll() - removes all the elements from the set

For example,

```
import java.util.HashSet;

class Main {
    public static void main(String[] args) {
        HashSet<Integer> numbers = new HashSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(6);
        System.out.println("HashSet: " + numbers);

        // Using remove() method
        boolean value1 = numbers.remove(5);
        System.out.println("Is 5 removed? " + value1);

        boolean value2 = numbers.removeAll(numbers);
        System.out.println("Are all elements removed? " + value2);
    }
}
```

Output

```
HashSet: [2, 5, 6]
Is 5 removed? true
Are all elements removed? true
```

Set Operations

The various methods of the HashSet class can also be used to perform various set operations.

Union of Sets

To perform the union between two sets, we can use the `addAll()` method. For example,

```
import java.util.HashSet;

class Main {
    public static void main(String[] args) {
        HashSet<Integer> evenNumbers = new HashSet<>();
        evenNumbers.add(2);
        evenNumbers.add(4);
        System.out.println("HashSet1: " + evenNumbers);

        HashSet<Integer> numbers = new HashSet<>();
        numbers.add(1);
        numbers.add(3);
        System.out.println("HashSet2: " + numbers);

        // Union of two set
        numbers.addAll(evenNumbers);
        System.out.println("Union is: " + numbers);
    }
}
```

Output

```
HashSet1: [2, 4]
HashSet2: [1, 3]
Union is: [1, 2, 3, 4]
```

Intersection of Sets

To perform the intersection between two sets, we can use the retainAll() method. For example

```
import java.util.HashSet;

class Main {
    public static void main(String[] args) {
        HashSet<Integer> primeNumbers = new HashSet<>();
        primeNumbers.add(2);
        primeNumbers.add(3);
        System.out.println("HashSet1: " + primeNumbers);

        HashSet<Integer> evenNumbers = new HashSet<>();
        evenNumbers.add(2);
        evenNumbers.add(4);
        System.out.println("HashSet2: " + evenNumbers);

        // Intersection of two sets
        evenNumbers.retainAll(primeNumbers);
        System.out.println("Intersection is: " + evenNumbers);
    }
}
```

Output

```
HashSet1: [2, 3]
HashSet2: [2, 4]
Intersection is: [2]
```

Difference of Sets

To calculate the difference between the two sets, we can use the `removeAll()` method. For example,

```
import java.util.HashSet;

class Main {
    public static void main(String[] args) {
        HashSet<Integer> primeNumbers = new HashSet<>();
        primeNumbers.add(2);
        primeNumbers.add(3);
        primeNumbers.add(5);
        System.out.println("HashSet1: " + primeNumbers);

        HashSet<Integer> oddNumbers = new HashSet<>();
        oddNumbers.add(1);
        oddNumbers.add(3);
        oddNumbers.add(5);
        System.out.println("HashSet2: " + oddNumbers);

        // Difference between HashSet1 and HashSet2
        primeNumbers.removeAll(oddNumbers);
        System.out.println("Difference : " + primeNumbers);
    }
}
```

Output

```
HashSet1: [2, 3, 5]
HashSet2: [1, 3, 5]
Difference: [2]
```

Subset

To check if a set is a subset of another set or not, we can use the containsAll() method. For example,

```
import java.util.HashSet;

class Main {
    public static void main(String[] args) {
        HashSet<Integer> numbers = new HashSet<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(4);
        System.out.println("HashSet1: " + numbers);
        HashSet<Integer> primeNumbers = new HashSet<>();
        primeNumbers.add(2);
        primeNumbers.add(3);
        System.out.println("HashSet2: " + primeNumbers);

        // Check if primeNumbers is a subset of numbers
        boolean result = numbers.containsAll(primeNumbers);
        System.out.println("Is HashSet2 is subset of HashSet1? " +
result);
    }
}
```

Output

```
HashSet1: [1, 2, 3, 4]
HashSet2: [2, 3]
Is HashSet2 is a subset of HashSet1? True
```

Other Methods Of HashSet

Method	Description
`clone()`	Creates a copy of the `HashSet`
`contains()`	Searches the `HashSet` for the specified element and returns a boolean result
`isEmpty()`	Checks if the `HashSet` is empty
`size()`	Returns the size of the `HashSet`
`clear()`	Removes all the elements from the `HashSet`

Why HashSet?

In Java, `HashSet` is commonly used if we have to access elements randomly. It is because elements in a hash table are accessed using hash codes.

The hashcode of an element is a unique identity that helps to identify the element in a hash table.

`HashSet` cannot contain duplicate elements. Hence, each hash set element has a unique hashcode.

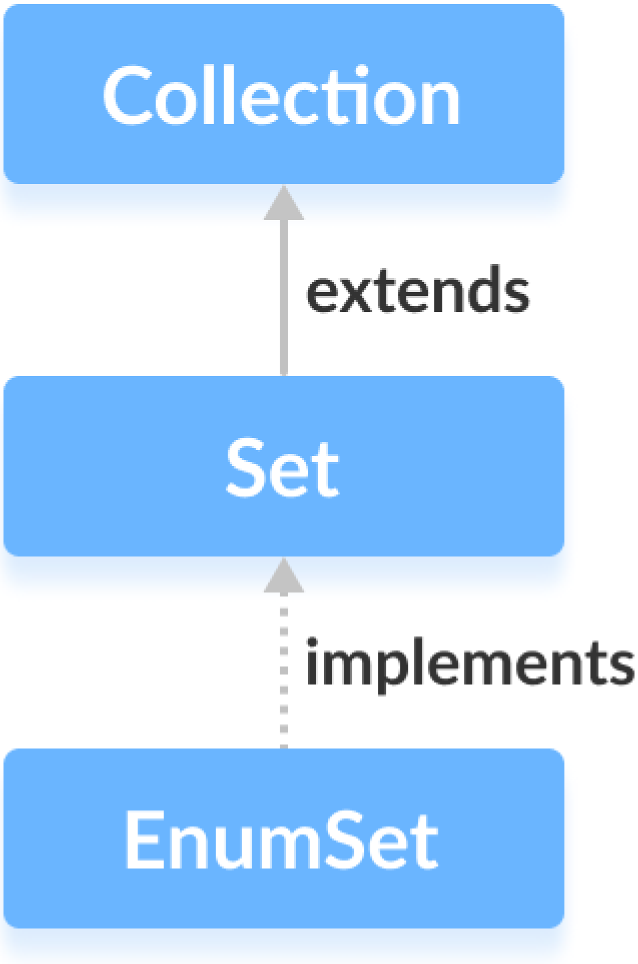
Note: HashSet is not synchronized. That is if multiple threads access the hash set at the same time and one of the threads modifies the hash set. Then it must be externally synchronized.

Java EnumSet

The EnumSet class of the Java collections framework provides a set implementation of elements of a single enum.

Before you learn about EnumSet, make sure to know about Java Enums.

It implements the Set interface.



Creating EnumSet

In order to create an enum set, we must import the `java.util.EnumSet` package first.

Unlike other set implementations, the enum set does not have public constructors. We must use the predefined methods to create an enum set.

1. Using allOf(Size)

The allof() method creates an enum set that contains all the values of the specified enum type *Size*. For example,

```
import java.util.EnumSet;

class Main {
    // an enum named Size
    enum Size {
        SMALL, MEDIUM, LARGE, EXTRALARGE
    }

    public static void main(String[] args) {

        // Creating an EnumSet using allOf()
        EnumSet<Size> sizes = EnumSet.allOf(Size.class);

        System.out.println("EnumSet: " + sizes);
    }

}
```

Output

```
EnumSet: [SMALL, MEDIUM, LARGE, EXTRALARGE]
```

Notice the statement,

```
EnumSet<Size> sizes = EnumSet.allOf(Size.class);
```

Here, *Size.class* denotes the *Size* enum that we have created.

2. Using noneOf(Size)

The noneOf() method creates an empty enum set. For example,

```
import java.util.EnumSet;

class Main {

     // an enum type Size
    enum Size {
        SMALL, MEDIUM, LARGE, EXTRALARGE
    }

    public static void main(String[] args) {

        // Creating an EnumSet using noneOf()
        EnumSet<Size> sizes = EnumSet.noneOf(Size.class);

        System.out.println("Empty EnumSet: " + sizes);
    }
}
```

Output

```
Empty EnumSet : []
```

Here, we have created an empty enum named *sizes*.

Note: We can only insert elements of enum type *Size* in the above program. It's because we created our empty enum set using *Size* enum.

3. Using range(e1, e2) Method

The range() method creates an enum set containing all the values of an enum between *e1* and *e2* including both values. For example,

```
import java.util.EnumSet;

class Main {

    enum Size {
        SMALL, MEDIUM, LARGE, EXTRALARGE
    }

    public static void main(String[] args) {

        // Creating an EnumSet using range()
        EnumSet<Size> sizes = EnumSet.range(Size.MEDIUM,
Size.EXTRALARGE);

        System.out.println("EnumSet: " + sizes);
    }
}
```

Output

```
EnumSet: [MEDIUM, LARGE, EXTRALARGE]
```

4. Using of() Method

The of() method creates an enum set containing the specified elements. For example,

```
import java.util.EnumSet;

class Main {

    enum Size {
        SMALL, MEDIUM, LARGE, EXTRALARGE
    }

    public static void main(String[] args) {

        // Using of() with a single parameter
        EnumSet<Size> sizes1 = EnumSet.of(Size.MEDIUM);
        System.out.println("EnumSet1: " + sizes1);

        EnumSet<Size> sizes2 = EnumSet.of(Size.SMALL, Size.LARGE);
        System.out.println("EnumSet2: " + sizes2);
    }
}
```

Output

```
EnumSet1: [MEDIUM]
EnumSet2: [SMALL, LARGE]
```

Methods of EnumSet

The `EnumSet` class provides methods that allow us to perform various elements on the enum set.

Insert Elements to EnumSet

- **`add()`** - inserts specified enum values to the enum set
- **`addAll()`** inserts all the elements of the specified collection to the set

For example,

```
import java.util.EnumSet;

class Main {

    enum Size {
        SMALL, MEDIUM, LARGE, EXTRALARGE
    }

    public static void main(String[] args) {

        // Creating an EnumSet using allOf()
        EnumSet<Size> sizes1 = EnumSet.allOf(Size.class);

        // Creating an EnumSet using noneOf()
        EnumSet<Size> sizes2 = EnumSet.noneOf(Size.class);

        // Using add method
        sizes2.add(Size.MEDIUM);
        System.out.println("EnumSet Using add(): " + sizes2);

        // Using addAll() method
        sizes2.addAll(sizes1);
        System.out.println("EnumSet Using addAll(): " + sizes2);
    }
}
```

Output

```
EnumSet using add(): [MEDIUM]
EnumSet using addAll(): [SMALL, MEDIUM, LARGE, EXTRALARGE]
```

In the above example, we have used the addAll() method to insert all the elements of an enum set *sizes1* to an enum set *sizes2*.

It's also possible to insert elements from other collections such as ArrayList, LinkedList, etc. to an enum set using addAll(). However, all collections should be of the same enum type.

Access EnumSet Elements

To access elements of an enum set, we can use the iterator() method. In order to use this method, we must import the java.util.Iterator package. For example,

```
import java.util.EnumSet;
import java.util.Iterator;

class Main {

    enum Size {
        SMALL, MEDIUM, LARGE, EXTRALARGE
    }

    public static void main(String[] args) {

        // Creating an EnumSet using allOf()
        EnumSet<Size> sizes = EnumSet.allOf(Size.class);

        Iterator<Size> iterate = sizes.iterator();

        System.out.print("EnumSet: ");
        while(iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }
    }
}
```

Output

```
EnumSet: SMALL, MEDIUM, LARGE, EXTRALARGE,
```

Note:

- **hasNext()** returns true if there is a next element in the enum set
- **next()** returns the next element in the enum set

Remove EnumSet Elements

- **`remove()`** - removes the specified element from the enum set
- **`removeAll()`** - removes all the elements from the enum set

For example,

```
import java.util.EnumSet;

class Main {

    enum Size {
        SMALL, MEDIUM, LARGE, EXTRALARGE
    }

    public static void main(String[] args) {

        // Creating EnumSet using allOf()
        EnumSet<Size> sizes = EnumSet.allOf(Size.class);
        System.out.println("EnumSet: " + sizes);

        // Using remove()
        boolean value1 = sizes.remove(Size.MEDIUM);
        System.out.println("Is MEDIUM removed? " + value1);

        // Using removeAll()
        boolean value2 = sizes.removeAll(sizes);
        System.out.println("Are all elements removed? " + value2);
    }
}
```

Output

```
EnumSet: [SMALL, MEDIUM, LARGE, EXTRALARGE]
Is MEDIUM removed? true
Are all elements removed? True
```

Other Methods

Method	Description
`copyOf()`	Creates a copy of the `EnumSet`
`contains()`	Searches the `EnumSet` for the specified element and returns a boolean result
`isEmpty()`	Checks if the `EnumSet` is empty

`size()`	Returns the size of the `EnumSet`
`clear()`	Removes all the elements from the `EnumSet`

Clonable and Serializable Interfaces

The `EnumSet` class also implements `Cloneable` and `Serializable` interfaces.

Cloneable Interface

It allows the `EnumSet` class to make a copy of instances of the class.

Serializable Interface

Whenever Java objects need to be transmitted over a network, objects need to be converted into bits or bytes. This is because Java objects cannot be transmitted over the network.

The Serializable interface allows classes to be serialized. This means objects of the classes implementing Serializable can be converted into bits or bytes.

Why EnumSet?

The EnumSet provides an efficient way to store enum values than other set implementations (like HashSet, TreeSet).

An enum set only stores enum values of a specific enum. Hence, the JVM already knows all the possible values of the set.

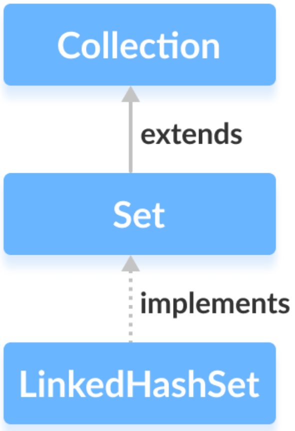
This is the reason why enum sets are internally implemented as a sequence of bits. Bits specifies whether elements are present in the enum set or not.

The bit of a corresponding element is turned on if that element is present in the set.

Java LinkedHashSet

The LinkedHashSet class of the Java collections framework provides functionalities of both the hashtable and the linked list data structure.

It implements the Set interface.



Elements of LinkedHashSet are stored in hash tables similar to HashSet.

However, linked hash sets maintain a doubly-linked list internally for all of its elements. The linked list defines the order in which elements are inserted in hash tables.

Create a LinkedHashSet

In order to create a linked hash set, we must import the java.util.LinkedHashSet package first.

Once we import the package, here is how we can create linked hash sets in Java.

```
// LinkedHashSet with 8 capacity and 0.75 load factor
LinkedHashSet<Integer> numbers = new LinkedHashSet<>(8, 0.75);
```

Here, we have created a linked hash set named *numbers*.

Notice, the part new LinkedHashSet<>(8, 0.75). Here, the first parameter is **capacity** and the second parameter is **loadFactor**.

- **capacity** - The capacity of this hash set is 8. Meaning, it can store 8 elements.
- **loadFactor** - The load factor of this hash set is 0.6. This means, whenever our hash table is filled by 60%, the elements are moved to a new hash table of double the size of the original hash table.

Default capacity and load factor

It's possible to create a linked hash set without defining its capacity and load factor. For example,

```
// LinkedHashSet with default capacity and load factor
LinkedHashSet<Integer> numbers1 = new LinkedHashSet<>();
```

By default,

- the capacity of the linked hash set will be 16
- the load factor will be 0.75

Creating LinkedHashSet from Other Collections

Here is how we can create a linked hash set containing all the elements of other collections.

```
import java.util.LinkedHashSet;
import java.util.ArrayList;

class Main {
    public static void main(String[] args) {
        // Creating an arrayList of even numbers
        ArrayList<Integer> evenNumbers = new ArrayList<>();
        evenNumbers.add(2);
        evenNumbers.add(4);
        System.out.println("ArrayList: " + evenNumbers);

        // Creating a LinkedHashSet from an ArrayList
        LinkedHashSet<Integer> numbers = new
LinkedHashSet<>(evenNumbers);
        System.out.println("LinkedHashSet: " + numbers);
    }
}
```

Output

```
ArrayList: [2, 4]
LinkedHashSet: [2, 4]
```

Methods of LinkedHashSet

The LinkedHashSet class provides methods that allow us to perform various operations on the linked hash set.

Insert Elements to LinkedHashSet

- **`add()`** - inserts the specified element to the linked hash set
- **`addAll()`** - inserts all the elements of the specified collection to the linked hash set

For example,

```
import java.util.LinkedHashSet;

class Main {
    public static void main(String[] args) {
        LinkedHashSet<Integer> evenNumber = new LinkedHashSet<>();

        // Using add() method
        evenNumber.add(2);
        evenNumber.add(4);
        evenNumber.add(6);
        System.out.println("LinkedHashSet: " + evenNumber);

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

        // Using addAll() method
        numbers.addAll(evenNumber);
        numbers.add(5);
        System.out.println("New LinkedHashSet: " + numbers);
    }
}
```

Output

```
LinkedHashSet: [2, 4, 6]
New LinkedHashSet: [2, 4, 6, 5]
```

Access LinkedHashSet Elements

To access the elements of a linked hash set, we can use the `iterator()` method. In order to use this method, we must import the `java.util.Iterator` package. For example,

```
import java.util.LinkedHashSet;
import java.util.Iterator;

class Main {
    public static void main(String[] args) {
        LinkedHashSet<Integer> numbers = new LinkedHashSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(6);
        System.out.println("LinkedHashSet: " + numbers);

        // Calling the iterator() method
        Iterator<Integer> iterate = numbers.iterator();

        System.out.print("LinkedHashSet using Iterator: ");

        // Accessing elements
        while(iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }
    }
}
```

Output

```
LinkedHashSet: [2, 5, 6]
LinkedHashSet using Iterator: 2, 5, 6,
```

Note:

- **`hasNext()`** returns `true` if there is a next element in the linked hash set
- **`next()`** returns the next element in the linked hash set

Remove Elements from HashSet

- **`remove()`** - removes the specified element from the linked hash set
- **`removeAll()`** - removes all the elements from the linked hash set

For example,

```
import java.util.LinkedHashSet;

class Main {
    public static void main(String[] args) {
        LinkedHashSet<Integer> numbers = new LinkedHashSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(6);
        System.out.println("LinkedHashSet: " + numbers);

        // Using the remove() method
        boolean value1 = numbers.remove(5);
        System.out.println("Is 5 removed? " + value1);

        boolean value2 = numbers.removeAll(numbers);
        System.out.println("Are all elements removed? " + value2);
    }
}
```

Output

```
LinkedHashSet: [2, 5, 6]
Is 5 removed? true
Are all elements removed? True
```

Set Operations

The various methods of the `LinkedHashSet` class can also be used to perform various set operations.

Union of Sets

Two perform the union between two sets, we can use the `addAll()` method. For example,

```
import java.util.LinkedHashSet;

class Main {
    public static void main(String[] args) {
        LinkedHashSet<Integer> evenNumbers = new LinkedHashSet<>();
        evenNumbers.add(2);
        evenNumbers.add(4);
        System.out.println("LinkedHashSet1: " + evenNumbers);

        LinkedHashSet<Integer> numbers = new LinkedHashSet<>();
        numbers.add(1);
        numbers.add(3);
        System.out.println("LinkedHashSet2: " + numbers);

        // Union of two set
        numbers.addAll(evenNumbers);
        System.out.println("Union is: " + numbers);
    }
}
```

Output

```
LinkedHashSet1: [2, 4]
LinkedHashSet2: [1, 3]
Union is: [1, 3, 2, 4]
```

Intersection of Sets

To perform the intersection between two sets, we can use the `retainAll()` method. For example

```
import java.util.LinkedHashSet;

class Main {
    public static void main(String[] args) {
        LinkedHashSet<Integer> primeNumbers = new LinkedHashSet<>();
        primeNumbers.add(2);
        primeNumbers.add(3);
        System.out.println("LinkedHashSet1: " + primeNumbers);

        LinkedHashSet<Integer> evenNumbers = new LinkedHashSet<>();
        evenNumbers.add(2);
        evenNumbers.add(4);
        System.out.println("LinkedHashSet2: " + evenNumbers);

        // Intersection of two sets
        evenNumbers.retainAll(primeNumbers);
        System.out.println("Intersection is: " + evenNumbers);
    }
}
```

Output

```
LinkedHashSet1: [2, 3]
LinkedHashSet2: [2, 4]
Intersection is: [2]
```

Difference of Sets

To calculate the difference between the two sets, we can use the `removeAll()` method. For example,

```
import java.util.LinkedHashSet;

class Main {
    public static void main(String[] args) {
        LinkedHashSet<Integer> primeNumbers = new LinkedHashSet<>();
        primeNumbers.add(2);
        primeNumbers.add(3);
        primeNumbers.add(5);
        System.out.println("LinkedHashSet1: " + primeNumbers);

        LinkedHashSet<Integer> oddNumbers = new LinkedHashSet<>();
        oddNumbers.add(1);
        oddNumbers.add(3);
        oddNumbers.add(5);
        System.out.println("LinkedHashSet2: " + oddNumbers);

        // Difference between LinkedHashSet1 and LinkedHashSet2
        primeNumbers.removeAll(oddNumbers);
        System.out.println("Difference : " + primeNumbers);
    }
}
```

Output

```
LinkedHashSet1: [2, 3, 5]
LinkedHashSet2: [1, 3, 5]
Difference: [2]
```

Subset

To check if a set is a subset of another set or not, we can use the `containsAll()` method. For example,

```
import java.util.LinkedHashSet;

class Main {
    public static void main(String[] args) {
        LinkedHashSet<Integer> numbers = new LinkedHashSet<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(4);
        System.out.println("LinkedHashSet1: " + numbers);

        LinkedHashSet<Integer> primeNumbers = new LinkedHashSet<>();
        primeNumbers.add(2);
        primeNumbers.add(3);
        System.out.println("LinkedHashSet2: " + primeNumbers);

        // Check if primeNumbers is a subset of numbers
        boolean result = numbers.containsAll(primeNumbers);
        System.out.println("Is LinkedHashSet2 is subset of
LinkedHashSet1? " + result);
```

```
    }
}
```

Output

```
LinkedHashSet1: [1, 2, 3, 4]
LinkedHashSet2: [2, 3]
Is LinkedHashSet2 is a subset of LinkedHashSet1? true
```

Other Methods Of LinkedHashSet

Method	Description
`clone()`	Creates a copy of the `LinkedHashSet`
`contains()`	Searches the `LinkedHashSet` for the specified element and returns a boolean result
`isEmpty()`	Checks if the `LinkedHashSet` is empty
`size()`	Returns the size of the `LinkedHashSet`
`clear()`	Removes all the elements from the `LinkedHashSet`

LinkedHashSet Vs. HashSet

Both LinkedHashSet and HashSet implements the Set interface. However, there exist some differences between them.

- LinkedHashSet maintains a linked list internally. Due to this, it maintains the insertion order of its elements.
- The LinkedHashSet class requires more storage than HashSet. This is because LinkedHashSet maintains linked lists internally.
- The performance of LinkedHashSet is slower than HashSet. It is because of linked lists present in LinkedHashSet.

LinkedHashSet Vs. TreeSet

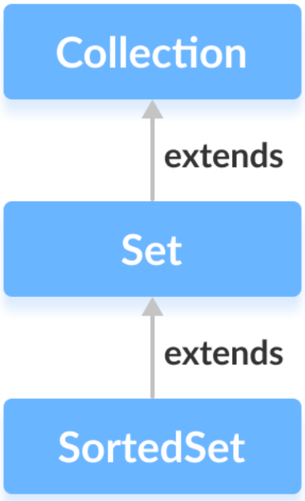
Here are the major differences between LinkedHashSet and TreeSet:

- The `TreeSet` class implements the `SortedSet` interface. That's why elements in a tree set are sorted. However, the `LinkedHashSet` class only maintains the insertion order of its elements.
- A `TreeSet` is usually slower than a `LinkedHashSet`. It is because whenever an element is added to a `TreeSet`, it has to perform the sorting operation.
- `LinkedHashSet` allows the insertion of null values. However, we cannot insert a null value to `TreeSet`.

Java SortedSet Interface

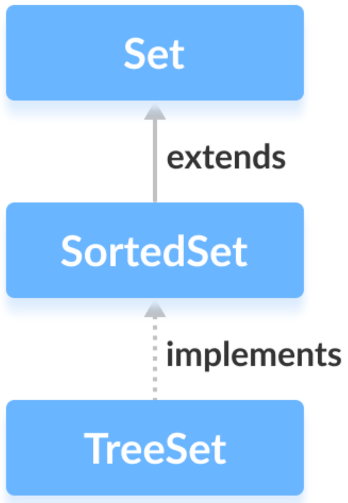
The SortedSet interface of the Java Collections framework is used to store elements with some order in a set.

It extends the Set interface.



Class that implements SortedSet

In order to use the functionalities of the SortedSet interface, we need to use the TreeSet class that implements it.



How to use SortedSet?

To use `SortedSet`, we must import the `java.util.SortedSet` package first.

```
// SortedSet implementation by TreeSet class
SortedSet<String> animals = new TreeSet<>();
```

We have created a sorted set called *animals* using the TreeSet class.

Here we have used no arguments to create a sorted set. Hence the set will be sorted naturally.

Methods of SortedSet

The SortedSet interface includes all the methods of the Set interface. It's because Set is a super interface of SortedSet.

Besides methods included in the Set interface, the SortedSet interface also includes these methods:

- **comparator()** - returns a comparator that can be used to order elements in the set
- **first()** - returns the first element of the set
- **last()** - returns the last element of the set
- **headSet(element)** - returns all the elements of the set before the specified element
- **tailSet(element)** - returns all the elements of the set after the specified element including the specified element
- **subSet(element1, element2)** - returns all the elements between the *element1* and *element2* including *element1*

Implementation of SortedSet in TreeSet Class

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

class Main {

    public static void main(String[] args) {
        // Creating SortedSet using the TreeSet
        SortedSet<Integer> numbers = new TreeSet<>();

        // Insert elements to the set
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(4);
        System.out.println("SortedSet: " + numbers);

        // Access the element
        int firstNumber = numbers.first();
        System.out.println("First Number: " + firstNumber);

        int lastNumber = numbers.last();
        System.out.println("Last Number: " + lastNumber);

        // Remove elements
        boolean result = numbers.remove(2);
        System.out.println("Is the number 2 removed? " + result);
    }
}
```

Output

```
SortedSet: [1, 2, 3, 4]
First Number: 1
Last Number: 4
Is the number 2 removed? true
```

Now that we know about the `SortedSet` interface, we will learn about its implementation using the `TreeSet` class.

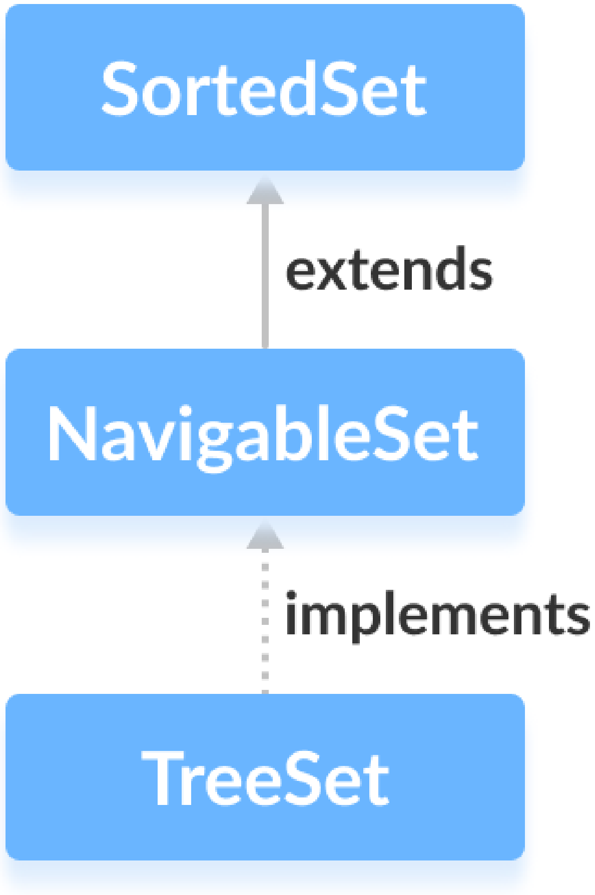
Java NavigableSet Interface

The NavigableSet interface of the Java Collections framework provides the features to navigate among the set elements.

It is considered as a type of SortedSet.

Class that implements NavigableSet

In order to use the functionalities of the NavigableSet interface, we need to use the TreeSet class that implements NavigableSet.



How to use NavigableSet?

In Java, we must import the `java.util.NavigableSet` package to use `NavigableSet`. Once we import the package, here's how we can create navigable sets.

```
// SortedSet implementation by TreeSet class
NavigableSet<String> numbers = new TreeSet<>();
```

Here, we have created a navigable set named *numbers* of the TreeSet class.

Methods of NavigableSet

The NavigableSet is considered as a type of SortedSet. It is because NavigableSet extends the SortedSet interface.

Hence, all *SortedSet* methods are also available in *NavigableSet*. To learn how these methods, visit Java SortedSet.

However, some of the methods of SortedSet (headSet(), tailSet() and subSet()) are defined differently in NavigableSet.

Let's see how these methods are defined in NavigableSet.

headSet(element, booleanValue)

The headSet() method returns all the elements of a navigable set before the specified *element* (which is passed as an argument).

The *booleanValue* parameter is optional. Its default value is false.

If true is passed as a *booleanValue,* the method returns all the elements before the specified element including the specified element.

tailSet(element, booleanValue)

The `tailSet()` method returns all the elements of a navigable set after the specified *element* (which is passed as an argument) including the specified element.

The *booleanValue* parameter is optional. Its default value is `true`.

If `false` is passed as a *booleanValue*, the method returns all the elements after the specified element without including the specified element.

subSet(e1, bv1, e2, bv2)

The subSet() method returns all the elements between *e1* and *e2* including *e1*.

The *bv1* and *bv2* are optional parameters. The default value of *bv1* is true, and the default value of *bv2* is false.

If false is passed as *bv1*, the method returns all the elements between *e1* and *e2* without including e1.

If true is passed as *bv2*, the method returns all the elements between *e1* and *e2*, including *e1*.

Methods for Navigation

The NavigableSet provides various methods that can be used to navigate over its elements.

- **descendingSet()** - reverses the order of elements in a set
- **descendingIterator()** - returns an iterator that can be used to iterate over a set in reverse order
- **ceiling()** - returns the lowest element among those elements that are greater than or equal to the specified element
- **floor()** - returns the greatest element among those elements that are less than or equal to the specified element
- **higher()** - returns the lowest element among those elements that are greater than the specified element
- **lower()** - returns the greatest element among those elements that are less than the specified element
- **pollFirst()** - returns and removes the first element from the set
- **pollLast()** - returns and removes the last element from the set

Implementation of NavigableSet in TreeSet Class

```
import java.util.NavigableSet;
import java.util.TreeSet;

class Main {

    public static void main(String[] args) {
        // Creating NavigableSet using the TreeSet
        NavigableSet<Integer> numbers = new TreeSet<>();

        // Insert elements to the set
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        System.out.println("NavigableSet: " + numbers);

        // Access the first element
        int firstElement = numbers.first();
        System.out.println("First Number: " + firstElement);

        // Access the last element
        int lastElement = numbers.last();
        System.out.println("Last Element: " + lastElement);

        // Remove the first element
        int number1 = numbers.pollFirst();
        System.out.println("Removed First Element: " + number1);

        // Remove the last element
        int number2 = numbers.pollLast();
        System.out.println("Removed Last Element: " + number2);

    }
}
```

Output

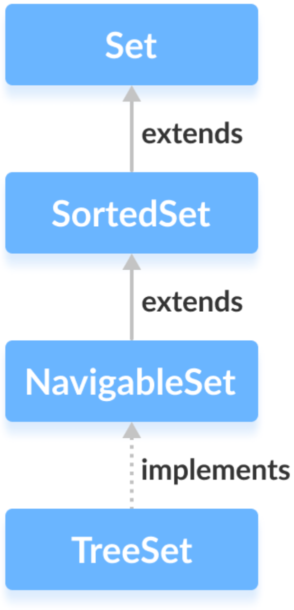
```
NavigableSet: [1, 2, 3]
First Element: 1
Last Element: 3
Removed First Element: 1
Removed Last Element: 3
```

Now that we know about the `NavigableSet` interface, we will learn about its implementation using the `TreeSet` class.

Java TreeSet

The TreeSet class of the Java collections framework provides the functionality of a tree data structure.

It extends the NavigableSet interface.



Creating a TreeSet

In order to create a tree set, we must import the java.util.TreeSet package first.

Once we import the package, here is how we can create a TreeSet in Java.

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

Here, we have created a TreeSet without any arguments. In this case, the elements in TreeSet are sorted naturally (ascending order).

However, we can customize the sorting of elements by using the Comparator interface.

Methods of TreeSet

The `TreeSet` class provides various methods that allow us to perform various operations on the set.

Insert Elements to TreeSet

- **`add()`** - inserts the specified element to the set
- **`addAll()`** - inserts all the elements of the specified collection to the set

For example,

```
import java.util.TreeSet;

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

        TreeSet<Integer> evenNumbers = new TreeSet<>();

        // Using the add() method
        evenNumbers.add(2);
        evenNumbers.add(4);
        evenNumbers.add(6);
        System.out.println("TreeSet: " + evenNumbers);

        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(1);

        // Using the addAll() method
        numbers.addAll(evenNumbers);
        System.out.println("New TreeSet: " + numbers);
    }
}
```

Output

```
TreeSet: [2, 4, 6]
New TreeSet: [1, 2, 4, 6]
```

Access TreeSet Elements

To access the elements of a tree set, we can use the `iterator()` method. In order to use this method, we must import `java.util.Iterator` package. For example,

```
import java.util.TreeSet;
import java.util.Iterator;
```

```
class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(6);
        System.out.println("TreeSet: " + numbers);

        // Calling iterator() method
        Iterator<Integer> iterate = numbers.iterator();
        System.out.print("TreeSet using Iterator: ");
        // Accessing elements
        while(iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }
    }
}
```

Output

```
TreeSet: [2, 5, 6]
TreeSet using Iterator: 2, 5, 6,
```

Remove Elements

- **`remove()`** - removes the specified element from the set
- **`removeAll()`** - removes all the elements from the set

For example,

```
import java.util.TreeSet;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(6);
        System.out.println("TreeSet: " + numbers);

        // Using the remove() method
        boolean value1 = numbers.remove(5);
        System.out.println("Is 5 removed? " + value1);

        // Using the removeAll() method
        boolean value2 = numbers.removeAll(numbers);
        System.out.println("Are all elements removed? " + value2);
    }
}
```

Output

```
TreeSet: [2, 5, 6]
Is 5 removed? true
Are all elements removed? true
```

Methods for Navigation

Since the `TreeSet` class implements `NavigableSet`, it provides various methods to navigate over the elements of the tree set.

1. first() and last() Methods

- **`first()`** - returns the first element of the set
- **`last()`** - returns the last element of the set

For example,

```
import java.util.TreeSet;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(6);
        System.out.println("TreeSet: " + numbers);

        // Using the first() method
        int first = numbers.first();
        System.out.println("First Number: " + first);

        // Using the last() method
        int last = numbers.last();
        System.out.println("Last Number: " + last);
    }
}
```

Output

```
TreeSet: [2, 5, 6]
First Number: 2
Last Number: 6
```

2. ceiling(), floor(), higher() and lower() Methods

- **higher(element)** - Returns the lowest element among those elements that are greater than the specified `element`.
- **lower(element)** - Returns the greatest element among those elements that are less than the specified `element`.
- **ceiling(element)** - Returns the lowest element among those elements that are greater than the specified *element*. If the *element* passed exists in a tree set, it returns the `element` passed as an argument.
- **floor(element)** - Returns the greatest element among those elements that are less than the specified `element`. If the *element* passed exists in a tree set, it returns the `element` passed as an argument.

For example,

```
import java.util.TreeSet;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(4);
        numbers.add(6);
        System.out.println("TreeSet: " + numbers);

        // Using higher()
        System.out.println("Using higher: " + numbers.higher(4));

        // Using lower()
        System.out.println("Using lower: " + numbers.lower(4));

        // Using ceiling()
        System.out.println("Using ceiling: " + numbers.ceiling(4));

        // Using floor()
        System.out.println("Using floor: " + numbers.floor(3));

    }
}
```

Output

```
TreeSet: [2, 4, 5, 6]
Using higher: 5
Using lower: 2
Using ceiling: 4
Using floor: 2
```

3. pollfirst() and pollLast() Methods

- **`pollFirst()`** - returns and removes the first element from the set
- **`pollLast()`** - returns and removes the last element from the set

For example,

```
import java.util.TreeSet;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(4);
        numbers.add(6);
        System.out.println("TreeSet: " + numbers);

        // Using pollFirst()
        System.out.println("Removed First Element: " +
numbers.pollFirst());

        // Using pollLast()
        System.out.println("Removed Last Element: " +
numbers.pollLast());

        System.out.println("New TreeSet: " + numbers);
    }
}
```

Output

```
TreeSet: [2, 4, 5, 6]
Removed First Element: 2
Removed Last Element: 6
New TreeSet: [4, 5]
```

4. headSet(), tailSet() and subSet() Methods

headSet(element, booleanValue)

The `headSet()` method returns all the elements of a tree set before the specified *element* (which is passed as an argument).

The *booleanValue* parameter is optional. Its default value is `false`.

If `true` is passed as a *booleanValue*, the method returns all the elements before the specified element including the specified element.

For example,

```
import java.util.TreeSet;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(4);
        numbers.add(6);
        System.out.println("TreeSet: " + numbers);

        // Using headSet() with default boolean value
        System.out.println("Using headSet without boolean value: " +
numbers.headSet(5));

        // Using headSet() with specified boolean value
        System.out.println("Using headSet with boolean value: " +
numbers.headSet(5, true));
    }
}
```

Output

```
TreeSet: [2, 4, 5, 6]
Using headSet without boolean value: [2, 4]
Using headSet with boolean value: [2, 4, 5]
```

tailSet(element, booleanValue)

The `tailSet()` method returns all the elements of a tree set after the specified *element* (which is passed as a parameter) including the specified *element*.

The *booleanValue* parameter is optional. Its default value is `true`.

If `false` is passed as a *booleanValue*, the method returns all the elements after the specified *element* without including the specified *element*.

For example,

```
import java.util.TreeSet;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(4);
        numbers.add(6);
        System.out.println("TreeSet: " + numbers);

        // Using tailSet() with default boolean value
        System.out.println("Using tailSet without boolean value: " +
numbers.tailSet(4));

        // Using tailSet() with specified boolean value
        System.out.println("Using tailSet with boolean value: " +
numbers.tailSet(4, false));
    }
}
```

Output

```
TreeSet: [2, 4, 5, 6]
Using tailSet without boolean value: [4, 5, 6]
Using tailSet with boolean value: [5, 6]
```

subSet(e1, bv1, e2, bv2)

The `subSet()` method returns all the elements between *e1* and *e2* including *e1*.

The *bv1* and *bv2* are optional parameters. The default value of *bv1* is `true`, and the default value of *bv2* is `false`.

If `false` is passed as *bv1*, the method returns all the elements between *e1* and *e2* without including `e1`.

If `true` is passed as *bv2*, the method returns all the elements between *e1* and *e2*, including *e1*.

For example,

```
import java.util.TreeSet;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(4);
        numbers.add(6);
        System.out.println("TreeSet: " + numbers);

        // Using subSet() with default boolean value
        System.out.println("Using subSet without boolean value: " +
numbers.subSet(4, 6));

        // Using subSet() with specified boolean value
        System.out.println("Using subSet with boolean value: " +
numbers.subSet(4, false, 6, true));
    }
}
```

Output

```
TreeSet: [2, 4, 5, 6]
Using subSet without boolean value: [4, 5]
Using subSet with boolean value: [5, 6]
```

Set Operations

The methods of the `TreeSet` class can also be used to perform various set operations.

Union of Sets

To perform the union between two sets, we use the addAll() method. For example,

```
import java.util.TreeSet;;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> evenNumbers = new TreeSet<>();
        evenNumbers.add(2);
        evenNumbers.add(4);
        System.out.println("TreeSet1: " + evenNumbers);

        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        System.out.println("TreeSet2: " + numbers);

        // Union of two sets
        numbers.addAll(evenNumbers);
        System.out.println("Union is: " + numbers);

    }
}
```

Output

```
TreeSet1: [2, 4]
TreeSet2: [1, 2, 3]
Union is: [1, 2, 3, 4]
```

Intersection of Sets

To perform the intersection between two sets, we use the retainAll() method. For example,

```
import java.util.TreeSet;;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> evenNumbers = new TreeSet<>();
        evenNumbers.add(2);
        evenNumbers.add(4);
        System.out.println("TreeSet1: " + evenNumbers);

        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        System.out.println("TreeSet2: " + numbers);

        // Intersection of two sets
        numbers.retainAll(evenNumbers);
        System.out.println("Intersection is: " + numbers);
    }
}
```

Output

```
TreeSet1: [2, 4]
TreeSet2: [1, 2, 3]
Intersection is: [2]
```

Difference of Sets

To calculate the difference between the two sets, we can use the removeAll() method. For example,

```
import java.util.TreeSet;;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> evenNumbers = new TreeSet<>();
        evenNumbers.add(2);
        evenNumbers.add(4);
        System.out.println("TreeSet1: " + evenNumbers);

        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(4);
        System.out.println("TreeSet2: " + numbers);

        // Difference between two sets
        numbers.removeAll(evenNumbers);
        System.out.println("Difference is: " + numbers);
    }
}
```

Output

```
TreeSet1: [2, 4]
TreeSet2: [1, 2, 3, 4]
Difference is: [1, 3]
```

Subset of a Set

To check if a set is a subset of another set or not, we use the containsAll() method. For example,

```
import java.util.TreeSet;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(4);
        System.out.println("TreeSet1: " + numbers);

        TreeSet<Integer> primeNumbers = new TreeSet<>();
        primeNumbers.add(2);
        primeNumbers.add(3);
        System.out.println("TreeSet2: " + primeNumbers);

        // Check if primeNumbers is subset of numbers
        boolean result = numbers.containsAll(primeNumbers);
        System.out.println("Is TreeSet2 subset of TreeSet1? " +
result);
    }
}
```

Output

```
TreeSet1: [1, 2, 3, 4]
TreeSet2: [2, 3]
Is TreeSet2 subset of TreeSet1? True
```

Other Methods of TreeSet

Method	Description
`clone()`	Creates a copy of the TreeSet
`contains()`	Searches the TreeSet for the specified element and returns a boolean result
`isEmpty()`	Checks if the `TreeSet` is empty
`size()`	Returns the size of the `TreeSet`
`clear()`	Removes all the elements from the `TreeSet`

TreeSet Vs. HashSet

Both the TreeSet as well as the HashSet implements the Set interface. However, there exist some differences between them.

- Unlike `HashSet`, elements in `TreeSet` are stored in some order. It is because `TreeSet` implements the `SortedSet` interface as well.
- `TreeSet` provides some methods for easy navigation. For example, `first()`, `last()`, `headSet(`), `tailSet()`, etc. It is because `TreeSet` also implements the `NavigableSet` interface.
- `HashSet` is faster than the `TreeSet` for basic operations like add, remove, contains and size.

TreeSet Comparator

In all the examples above, tree set elements are sorted naturally. However, we can also customize the ordering of elements.

For this, we need to create our own comparator class based on which elements in a tree set are sorted. For example,

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

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

        // Creating a tree set with customized comparator
        TreeSet<String> animals = new TreeSet<>(new
CustomComparator());

        animals.add("Dog");
        animals.add("Zebra");
        animals.add("Cat");
        animals.add("Horse");
        System.out.println("TreeSet: " + animals);
    }

    // Creating a comparator class
    public static class CustomComparator implements Comparator<String>
{

        @Override
        public int compare(String animal1, String animal2) {
            int value =  animal1.compareTo(animal2);

            // elements are sorted in reverse order
            if (value > 0) {
                return -1;
            }
            else if (value < 0) {
                return 1;
            }
            else {
                return 0;
            }
        }
    }
}
```

Output

```
TreeSet: [Zebra, Horse, Dog, Cat]
```

In the above example, we have created a tree set passing CustomComparator class as an argument.

The CustomComparator class implements the Comparator interface.

We then override the compare() method. The method will now sort elements in reverse order.

Java Algorithms

The Java collections framework provides various algorithms that can be used to manipulate elements stored in data structures.

Algorithms in Java are static methods that can be used to perform various operations on collections.

Since algorithms can be used on various collections, these are also known as **generic algorithms**.

Let's see the implementation of different methods available in the collections framework.

1. Sorting Using sort()

The sort() method provided by the collections framework is used to sort elements. For example,

```
import java.util.ArrayList;
import java.util.Collections;

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

        // Creating an array list
        ArrayList<Integer> numbers = new ArrayList<>();

        // Add elements
        numbers.add(4);
        numbers.add(2);
        numbers.add(3);
        System.out.println("Unsorted ArrayList: " + numbers);

        // Using the sort() method
        Collections.sort(numbers);
        System.out.println("Sorted ArrayList: " + numbers);

    }
}
```

Output

```
Unsorted ArrayList: [4, 2, 3]
Sorted ArrayList: [2, 3, 4]
```

Here the sorting occurs in natural order (ascending order). However, we can customize the sorting order of the sort() method using *the Comparator interface*.

2. Shuffling Using shuffle()

The shuffle() method of the Java collections framework is used to destroy any kind of order present in the data structure. It does just the opposite of the sorting. For example,

```
import java.util.ArrayList;
import java.util.Collections;

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

        // Creating an array list
        ArrayList<Integer> numbers = new ArrayList<>();

        // Add elements
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        System.out.println("Sorted ArrayList: " + numbers);

        // Using the shuffle() method
        Collections.shuffle(numbers);
        System.out.println("ArrayList using shuffle: " + numbers);

    }
}
```

Output

```
Sorted ArrayList: [1, 2, 3]
ArrayList using shuffle: [2, 1, 3]
```

When we run the program, the shuffle() method will return a random output.

The shuffling algorithm is mainly used in games where we want random output.

3. Routine Data Manipulation

In Java, the collections framework provides different methods that can be used to manipulate data.

- **`reverse()`** - reverses the order of elements
- **`fill()`** - replace every element in a collection with the specified value
- **`copy()`** - creates a copy of elements from the specified source to destination
- **`swap()`** - swaps the position of two elements in a collection
- **`addAll()`** - adds all the elements of a collection to other collection

For example,

```
import java.util.Collections;
import java.util.ArrayList;

class Main {
    public static void main(String[] args) {
        // Creating an ArrayList
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(2);
        System.out.println("ArrayList1: " + numbers);

        // Using reverse()
        Collections.reverse(numbers);
        System.out.println("Reversed ArrayList1: " + numbers);

        // Using swap()
        Collections.swap(numbers, 0, 1);
        System.out.println("ArrayList1 using swap(): " + numbers);

        ArrayList<Integer> newNumbers = new ArrayList<>();

        // Using addAll
        newNumbers.addAll(numbers);
        System.out.println("ArrayList2 using addAll(): " +
newNumbers);
```

```
        // Using fill()
        Collections.fill(numbers, 0);
        System.out.println("ArrayList1 using fill(): " + numbers);

        // Using copy()
        Collections.copy(newNumbers, numbers);
        System.out.println("ArrayList2 using copy(): " + newNumbers);
    }
}
```

Output

```
ArrayList1: [1, 2]
Reversed ArrayList1: [2, 1]
ArrayList1 Using swap(): [1, 2]
ArrayList2 using addALl(): [1, 2]
ArrayList1 using fill(): [0, 0]
ArrayList2 using copy(): [0, 0]
```

Note: While performing the `copy()` method both the lists should be of the same size.

4. Searching Using binarySearch()

The binarySearch() method of the Java collections framework searches for the specified element. It returns the position of the element in the specified collections. For example,

```
import java.util.Collections;
import java.util.ArrayList;

class Main {
    public static void main(String[] args) {
        // Creating an ArrayList
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);

        // Using binarySearch()
        int pos = Collections.binarySearch(numbers, 3);
        System.out.println("The position of 3 is " + pos);
    }
}
```

Output

```
The position of 3 is 2.
```

Note: The collection should be sorted before performing the binarySearch() method.

5. Composition

- **`frequency()`** - returns the count of the number of times an element is present in the collection
- **`disjoint()`** - checks if two collections contain some common element

For example,

```
import java.util.Collections;
import java.util.ArrayList;

class Main {
    public static void main(String[] args) {
        // Creating an ArrayList
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(2);
        System.out.println("ArrayList1: " + numbers);

        int count = Collections.frequency(numbers, 2);
        System.out.println("Count of 2: " + count);

        ArrayList<Integer> newNumbers = new ArrayList<>();
        newNumbers.add(5);
        newNumbers.add(6);
        System.out.println("ArrayList2: " + newNumbers);

        boolean value = Collections.disjoint(numbers, newNumbers);
        System.out.println("Two lists are disjoint: " + value);
    }
}
```

Output

```
ArrayList1: [1, 2, 3, 2]
Count of 2: 2
ArrayList2: [5, 6]
Two lists are disjoint: true
```

6. Finding Extreme Values

The `min()` and `max()` methods of the Java collections framework are used to find the minimum and the maximum elements, respectively. For example,

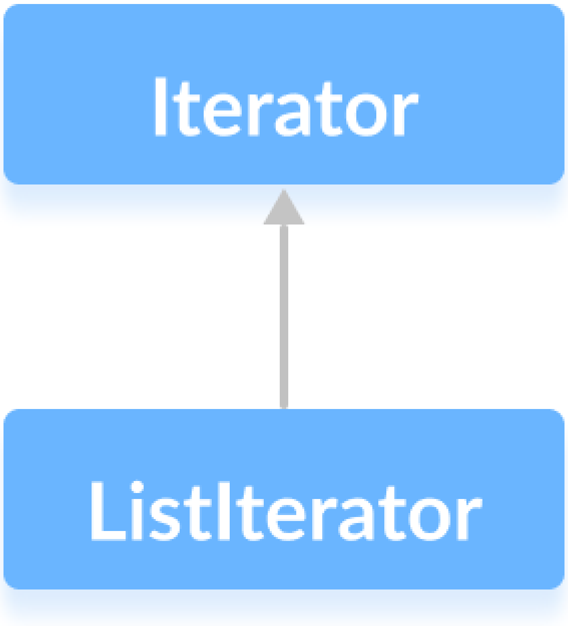
```
import java.util.Collections;
import java.util.ArrayList;

class Main {
    public static void main(String[] args) {
        // Creating an ArrayList
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);

        // Using min()
        int min = Collections.min(numbers);
        System.out.println("Minimum Element: " + min);

        // Using max()
        int max = Collections.max(numbers);
        System.out.println("Maximum Element: " + max);
    }
}
```

Output

```
Minimum Element: 1
Maximum Element: 3
```

Java Iterator Interface

The Iterator interface of the Java collections framework allows us to access elements of a collection. It has a subinterface ListIterator.



All the Java collections include an iterator() method. This method returns an instance of iterator used to iterate over elements of collections.

Methods of Iterator

The `Iterator` interface provides 4 methods that can be used to perform various operations on elements of collections.

- **`hasNext()`** - returns `true` if there exists an element in the collection
- **`next()`** - returns the next element of the collection
- **`remove()`** - removes the last element returned by the `next()`
- **`forEachRemaining()`** - performs the specified action for each remaining element of the collection

Example: Implementation of Iterator

In the example below, we have implemented the `hasNext()`, `next()`, `remove()` and `forEachRemining()` methods of the `Iterator` interface in an array list.

```
import java.util.ArrayList;
import java.util.Iterator;

class Main {
    public static void main(String[] args) {
        // Creating an ArrayList
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(3);
        numbers.add(2);
        System.out.println("ArrayList: " + numbers);

        // Creating an instance of Iterator
        Iterator<Integer> iterate = numbers.iterator();

        // Using the next() method
        int number = iterate.next();
        System.out.println("Accessed Element: " + number);

        // Using the remove() method
        iterate.remove();
        System.out.println("Removed Element: " + number);

        System.out.print("Updated ArrayList: ");

        // Using the hasNext() method
        while(iterate.hasNext()) {
            // Using the forEachRemaining() method
            iterate.forEachRemaining((value) -> System.out.print(value
+ ", "));
        }
    }
}
```

Output

```
ArrayList: [1, 3, 2]
Acessed Element: 1
Removed Element: 1
Updated ArrayList: 3, 2,
```

In the above example, notice the statement:

```
iterate.forEachRemaining((value) -> System.put.print(value + ", "));
```

Here, we have passed the lambda expression as an argument of the forEachRemaining() method.

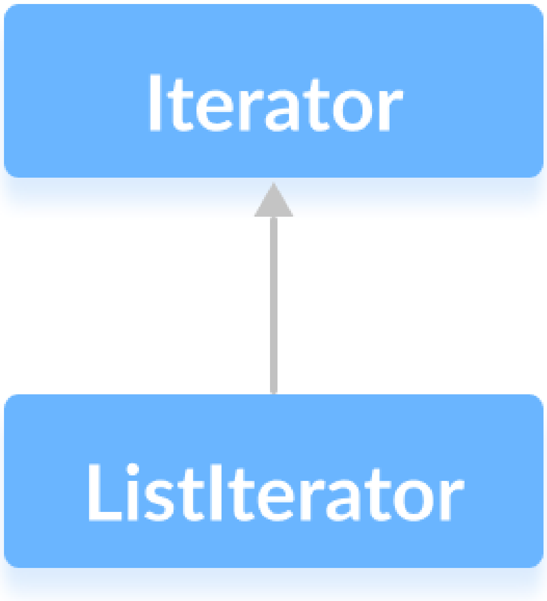
Now the method will print all the remaining elements of the array list.

Java ListIterator Interface

The ListIterator interface of the Java collections framework provides the functionality to access elements of a list.

It is bidirectional. This means it allows us to iterate elements of a list in both the direction.

It extends the Iterator interface.



The `List` interface provides a `listIterator()` method that returns an instance of the `ListIterator` interface.

Methods of ListIterator

The `ListIterator` interface provides methods that can be used to perform various operations on the elements of a list.

- `hasNext()` - returns true if there exists an element in the list
- `next()` - returns the next element of the list
- `nextIndex()` returns the index of the element that the `next()` method will return
- `previous()` - returns the previous element of the list
- `previousIndex()` - returns the index of the element that the `previous()` method will return
- `remove()` - removes the element returned by either `next()` or `previous()`
- `set()` - replaces the element returned by either `next()` or `previous()` with the specified element

Example 1: Implementation of ListIterator

In the example below, we have implemented the `next()`, `nextIndex()` and `hasNext()` methods of the `ListIterator` interface in an array list.

```
import java.util.ArrayList;
import java.util.ListIterator;

class Main {
    public static void main(String[] args) {
        // Creating an ArrayList
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(3);
        numbers.add(2);
        System.out.println("ArrayList: " + numbers);

        // Creating an instance of ListIterator
        ListIterator<Integer> iterate = numbers.listIterator();

        // Using the next() method
        int number1 = iterate.next();
        System.out.println("Next Element: " + number1);

        // Using the nextIndex()
        int index1 = iterate.nextIndex();
        System.out.println("Position of Next Element: " + index1);

        // Using the hasNext() method
        System.out.println("Is there any next element? " +
iterate.hasNext());
```

```
    }
}
```

Output

```
ArrayList: [1, 3, 2]
Next Element: 1
Position of Next Element: 1
Is there any next element? true
```

Example 2: Implementation of ListIterator

In the example below, we have implemented the `previous()` and `previousIndex()` methods of the `ListIterator` interface in an array list.

```
import java.util.ArrayList;
import java.util.ListIterator;

class Main {
    public static void main(String[] args) {
        // Creating an ArrayList
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(3);
        numbers.add(2);
        System.out.println("ArrayList: " + numbers);

        // Creating an instance of ListIterator
        ListIterator<Integer> iterate = numbers.listIterator();
        iterate.next();
        iterate.next();

        // Using the previous() method
        int number1 = iterate.previous();
        System.out.println("Previous Element: " + number1);

        // Using the previousIndex()
        int index1 = iterate.previousIndex();
        System.out.println("Position of the Previous element: " +
index1);
    }
}
```

Output

```
ArrayList: [1, 3, 2]
Previous Element: 3
Position of the Previous Element: 0
```

In the above example, initially, the instance of the `Iterator` was before 1. Since there was no element before 1 so calling the `previous()` method will throw an exception.

We then used the `next()` methods 2 times. Now the `Iterator` instance will be between 3 and 2.

Hence, the `previous()` method returns 3.