

# Collections Framework

**The Collections Framework: Collections Overview, The Collection Interfaces - Collection, List, Set and SortedSet, The Collection Classes - ArrayList, LinkedList, HashSet, LinkedHashMap and TreeSet, Accessing a Collection via an Iterator.**

-----

## Collections Overview

### Collections in Java

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

**You can perform following activity using Java collection framework**

Java Collections can achieve all the operations that you perform on a data such as

- Add objects to collection
- Remove objects from collection
- Search for an object in collection
- Retrieve/get object from collection
- Iterate through the collection for business specific functionality.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes ([ArrayList](#), Vector, [LinkedList](#), [PriorityQueue](#), HashSet, LinkedHashMap, TreeSet).

### What is Collection in Java

A Collection represents a single unit of objects, i.e., a group.

**What is a framework in Java**

- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

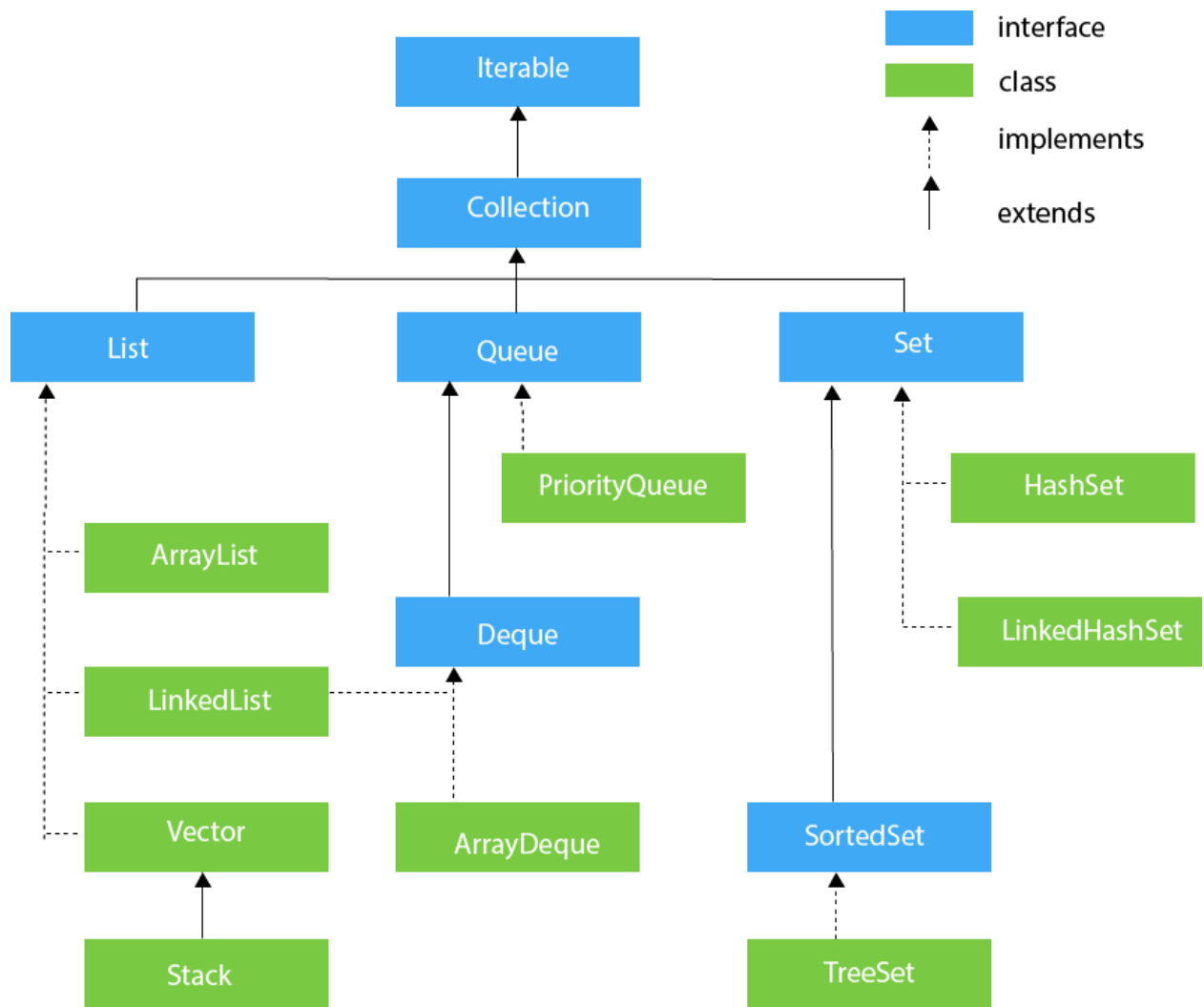
### What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

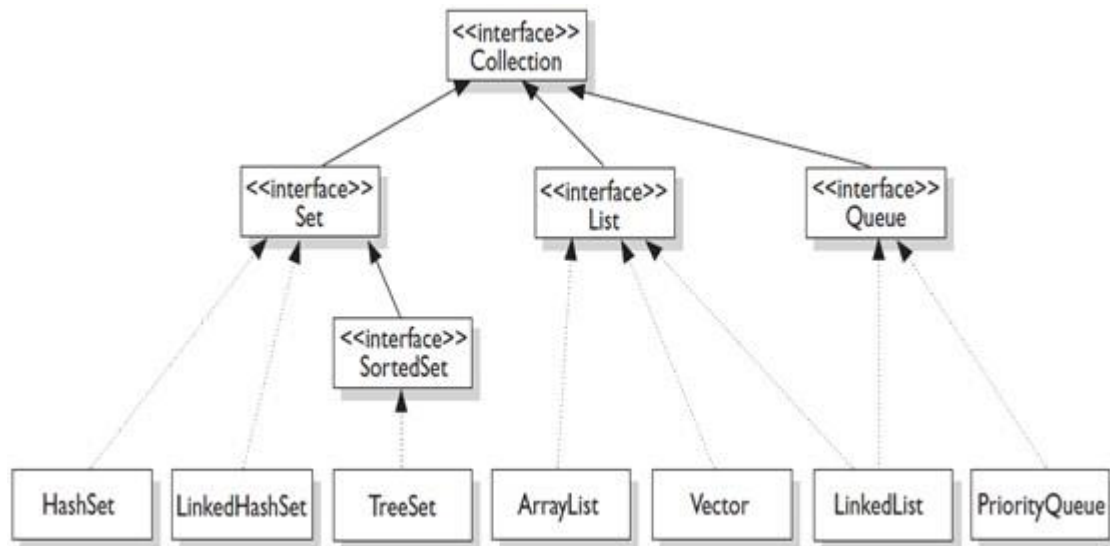
### Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The **java.util** package contains all the **classes** and **interfaces** for the Collection framework.

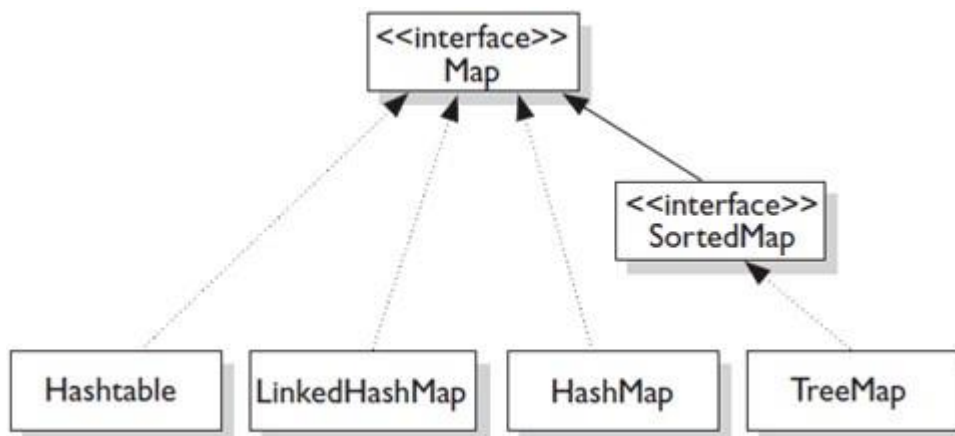


### Key Interfaces and classes of collection framework

- **collection** (lowercase c): It represents any of the data structures in which objects are stored and iterated over.
- **Collection** (capital C): It is actually the `java.util.Collection` interface from which `Set`, `List`, and `Queue` extend.
- **Collections** (capital C and ends with s): It is the `java.util.Collections` class that holds a pile of static utility methods for use with collections.



There are some other classes in collection framework which do not extend Collection Interface they implement Map interface.



## Interfaces involved in collections:

### 1.Iterator interface

Iterator interface provides the facility of iterating the elements in a forward direction only.

### Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.

3	public void remove()	It removes the last elements returned by the iterator. It is less
		used.

## 2.Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

### 1. Iterator<T> iterator()

It returns the iterator over the elements of type T.

## 3.Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add ( Object obj), Boolean addAll ( Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

## 4.List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack. To instantiate the List interface, we must use :

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. List <data-type> list4 = **new** Stack();
- 5.

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

## 5.Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

1. Queue<String> q1 = **new** PriorityQueue();
2. Queue<String> q2 = **new** ArrayDeque();

There are various classes that implement the Queue interface, some of them are given below.

### 6.Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

1. Deque d = **new** ArrayDeque();

### 7.Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

1. Set<data-type> s1 = **new** HashSet<data-type>();
2. Set<data-type> s2 = **new** LinkedHashSet<data-type>();
3. Set<data-type> s3 = **new** TreeSet<data-type>();

### 8.SortedSet Interface

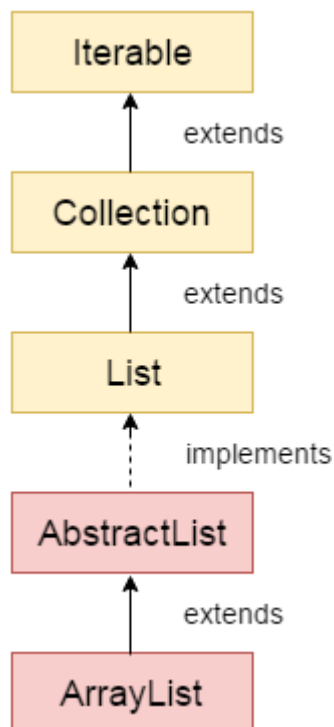
SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements. The SortedSet can be instantiated as:

1. SortedSet<data-type> set = **new** TreeSet();

## **The Collection Classes - ArrayList, LinkedList, HashSet, LinkedHashSet and TreeSet, Accessing a Collection via an Iterator.**

### **1,Java ArrayList:**

- ✓ Java provides an ArrayList class which can be used to create containers that store lists of objects.
- ✓ ArrayList can be considered as a growable array.
- ✓ Arraylist is an ordered collection (by index), but not sorted.



Java **ArrayList** class uses a *dynamic array* for storing the elements. It is like an array, but there is *no size limit*. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the *java.util* package. It is like the Vector in C++.

The ArrayList in Java can have the duplicate elements also. It implements the List interface so we can use all the methods of List interface here. The ArrayList maintains the insertion order internally.

It inherits the AbstractList class and implements **List interface**.

**The important points about Java ArrayList class are:**

- Java ArrayList class can

- contain duplicate elements. ○ Java ArrayList class maintains insertion

- order. ○ Java ArrayList class is non **synchronized**. ○ Java ArrayList

- allows random access because array works at the index basis.

- In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list

**To use the ArrayList class, you must use the following import statement:**

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

**Then, to declare an ArrayList, you can use the default constructor, as in the following example:**

```
ArrayList names = new ArrayList();
```

ArrayList methods	
Method	Purpose
public void add(Object) public void add(int, Object)	Adds an item to an ArrayList. The default version adds an item at the next available location; an overloaded version allows you to specify a position at which to add the item
public void remove(int)	Removes an item from an ArrayList at a specified location
public void set(int, Object)	Alters an item at a specified ArrayList location
Object get(int)	Retrieves an item from a specified location in an ArrayList
size()	Returns the current ArrayList size

Java Program to demonstrate the use of all above methods described above. Here we are creating ArrayList named myList and adding objects using add() method as well as using index based add method, then printing all the objects using for loop. Then there we demonstrate use of get(), contains(), and size() methods. the output of program is shown below the java code.

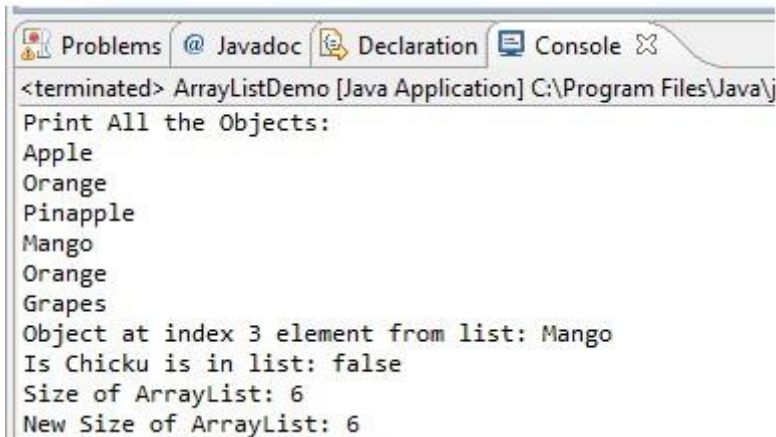
```
import java.util.ArrayList; public
class ArrayListDemo {
public static void main(String[] args) {
    //declaring ArrayList of String objects
    ArrayList<String> myList = new ArrayList<String>();
    //Adding objects to Array List at default index
    myList.add("Apple");
    myList.add("Mango");
    myList.add("Orange");
    myList.add("Grapes");
    //Adding object at specific index  myList.add(1, "Orange");
    myList.add(2, "Pinapple");
    System.out.println("Print All the Objects:");
    for(String s:myList){
        System.out.println(s);
    }
    System.out.println("Object at index 3 element from list: "+ myList.get(3));
    System.out.println("Is Chicku is in list: " + myList.contains("Chicku"));
    System.out.println("Size of ArrayList: " + myList.size());
    myList.remove("Papaya");
    System.out.println("New Size of ArrayList: "+ myList.size());
}
```

}



```
}
```

Output:



```
<terminated> ArrayListDemo [Java Application] C:\Program Files\Java\j
Print All the Objects:
Apple
Orange
Pinapple
Mango
Orange
Grapes
Object at index 3 element from list: Mango
Is Chicku is in list: false
Size of ArrayList: 6
New Size of ArrayList: 6
```

### Iterating ArrayList using Iterator

Let's see an example to traverse ArrayList elements using the Iterator interface.

```
import java.util.*;
public class ArrayListExample2{
public static void main(String args[]){
    ArrayList<String> list=new ArrayList<String>();//Creating
arraylist list.add("Mango");//Adding object in arraylist
list.add("Apple"); list.add("Banana"); list.add("Grapes");
    //Traversing list through Iterator
    Iterator itr=list.iterator();//getting the Iterator
    while(itr.hasNext()){//check if iterator has the elements
        System.out.println(itr.next());//printing the element and move to next
    }
}
}
```

Output:

```
Mango
Apple
Banana
Grapes
```

### Get and Set ArrayList

The *get()* method returns the element at the specified index, whereas the *set()* method changes the element.

```
import java.util.*;
public class ArrayListExample4{
    public static void main(String args[]){
        ArrayList<String> al=new ArrayList<String>();
        al.add("Mango");
        al.add("Apple");
        al.add("Banana");
        al.add("Grapes");
        //accessing the element
        System.out.println("Returning element: "+al.get(1));//it will return the 2nd element, because i
index starts from 0
        //changing the element
        al.set(1,"Dates");
        //Traversing list
        for(String fruit:al)
            System.out.println(fruit);
    }
}
```

#### Output:

Returning element: Apple  
Mango  
Dates  
Banana  
Grapes

#### How to Sort ArrayList

The *java.util* package provides a utility class **Collections** which has the static method `sort()`. Using the **Collections.sort()** method, we can easily sort the ArrayList.

```
import java.util.*; class
SortArrayList{
    public static void main(String args[]){
        //Creating a list of fruits
        List<String> list1=new
        ArrayList<String>(); list1.add("Mango");
        list1.add("Apple"); list1.add("Banana");
        list1.add("Grapes"); //Sorting the list
        Collections.sort(list1);
        //Traversing list through the for-each loop
        for(String fruit:list1)
        System.out.println(fruit);

        System.out.println("Sorting numbers...");
        //Creating a list of numbers
        List<Integer> list2=new ArrayList<Integer>();
```

Grapes  
Mango  
Sorting numbers...  
1  
11  
21  
51

### Summary of ArrayList:

- ArrayList and Vector are similar classes only difference is Vector has all method synchronized. Both class simple terms can be considered as a growable array.

```

list2.add(21);
list2.add(11);
list2.add(51);
list2.add(1);
//Sorting the list
Collections.sort(list2);
//Traversing list through the for-each loop
for(Integer number:list2)
    System.out.println(number);
}
}

```

#### **Output:**

Apple

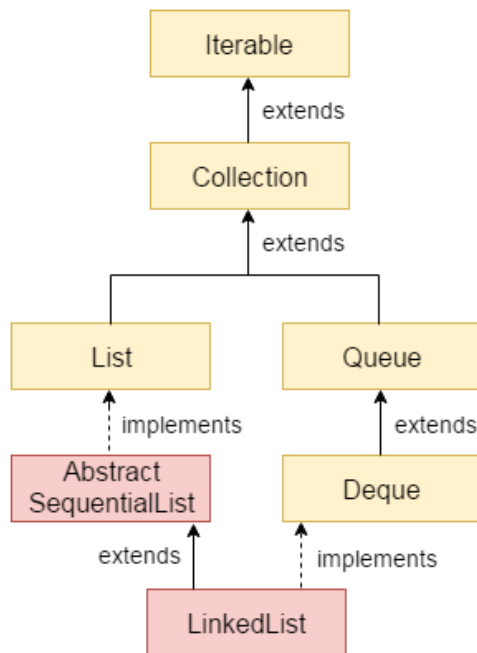
Banana

- ArrayList should be used in an application when we need to search objects from the list based on the index.
- ArrayList performance degrades when there is lots of insert and update operation in the middle of the list.

## **2.Java LinkedList class**

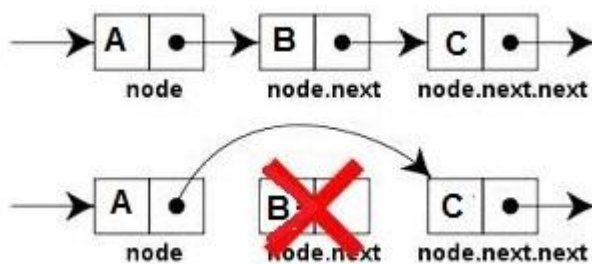
**The important points about Java LinkedList are:**

- Java LinkedList class can contain duplicate elements. ○ Java LinkedList class maintains insertion order. ○ Java LinkedList class is non synchronized. ○ In Java LinkedList class, manipulation is fast because no shifting needs to occur.
- Java LinkedList class can be used as a list, stack or queue.



Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

A LinkedList is ordered by index position, like ArrayList, except that the elements are doublylinked to one another. This linkage gives you new methods (beyond what you get from the List interface) for adding and removing from the beginning or end, which makes it an easy choice for implementing a stack or queue. Linked list has a concept of nodes and data. Here Node is storing values of next node while data stores the value it is holding. Below diagram shows how LinkedList storing values. There are three elements in LinkedList A, B and C. We are removing element B from the middle of the LinkedList which will just change node value of element A's node to point to node C.



Keep in mind that a LinkedList may iterate more slowly than an ArrayList, but it's a good choice when you need fast insertion and deletion.

### Constructors of Java LinkedList

Constructor	Description
LinkedList()	It is used to construct an empty list.
LinkedList(Collection<? extends E> c)	It is used to construct a list containing the elements of the specified collection, in the order, they are returned by the collection's iterator.

### Important methods of LinkedList class:

Method	Description
addFirst() or offerFirst( )	To add elements to the start of a list
addLast( ) or offerLast( ) or add()	To add elements to the end of the list
getFirst( ) or peekFirst( )	To obtain the first element of the list
getLast( ) or peekLast( )	To obtain the last element of the list
removeFirst( ) or pollFirst( ) or remove()	To remove the first element of the list
removeLast( ) or pollLast( )	To remove the last element of the list

Java Program to demonstrate use of all above methods described above. Here we are creating LinkedList named myLinkedList and adding objects using add(), addFirst() and addLast() methods as well as using index based add() method, then printing all the objects. Then modifying the list using remove(), removeLast() and remove(Object o) methods. Then we demonstrate use of getFirst() and getLast() methods. Output of program is shown below the java code.

#### Example1:

```
import java.util.LinkedList;
```

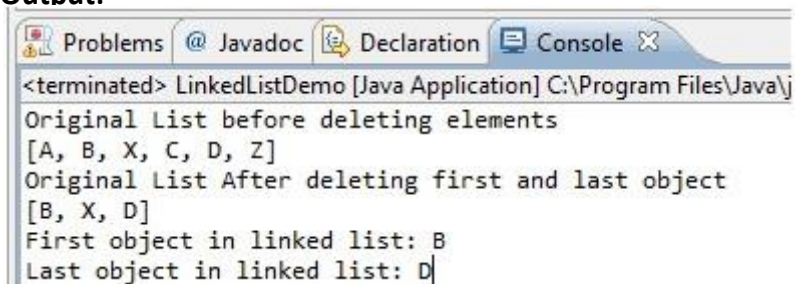
```
public class LinkedListDemo {  
    public static void main(String[] args) {  
        LinkedList<String> myLinkedList = new LinkedList<String>();  
        myLinkedList.addFirst("A");  
        myLinkedList.add("B");  
        myLinkedList.add("C");  
        myLinkedList.add("D");  
        myLinkedList.add(2, "X");//This will add C at index 2  
        myLinkedList.addLast("Z");  
        System.out.println("Original List before deleting elements");  
        System.out.println(myLinkedList);  
    }  
}
```

```

        myLinkedList.remove();
        myLinkedList.removeLast();
        myLinkedList.remove("C");
        System.out.println("Original List After deleting first and last object");
        System.out.println(myLinkedList);
        System.out.println("First object in linked list: "+ myLinkedList.getFirst());
        System.out.println("Last object in linked list: "+ myLinkedList.peekLast());
    }
}

```

### Output:



```

<terminated> LinkedListDemo [Java Application] C:\Program Files\Java\j
Original List before deleting elements
[A, B, X, C, D, Z]
Original List After deleting first and last object
[B, X, D]
First object in linked list: B
Last object in linked list: D

```

### Example2:Java LinkedList Example to reverse a list of elements

```

import java.util.*; public class
LinkedList4{ public static void
main(String args[]){

    LinkedList<String> ll=new
LinkedList<String>();      ll.add("Ravi");
ll.add("Vijay");          ll.add("Ajay");
    //Traversing the list of elements in reverse order
    Iterator i=ll.descendingIterator();    while(i.hasNext())
    {
        System.out.println(i.next());
    }
}
}

```

Output: Ajay

Vijay

Ravi

**Example3:Java LinkedList example to remove elements** Here,  
we see different ways to remove an element.

```
import java.util.*; public
class LinkedList3 {

    public static void main(String [] args)
    {
        LinkedList<String> ll=new
LinkedList<String>();      ll.add("Ravi");
ll.add("Vijay");          ll.add("Ajay");
ll.add("Anuj");           ll.add("Gaurav");
ll.add("Harsh");          ll.add("Virat");
ll.add("Gaurav");         ll.add("Harsh");
ll.add("Amit");

        System.out.println("Initial list of elements: "+ll);
//Removing specific element from arraylist
ll.remove("Vijay");

        System.out.println("After invoking remove(object) method: "+ll);
//Removing element on the basis of specific position
ll.remove(0);

        System.out.println("After invoking remove(index) method:
"+ll);      LinkedList<String> ll2=new LinkedList<String>();
ll2.add("Ravi");          ll2.add("Hanumat");
// Adding new elements to arraylist
ll.addAll(ll2);

        System.out.println("Updated list : "+ll);
//Removing all the new elements from arraylist
ll.removeAll(ll2);

        System.out.println("After invoking removeAll() method: "+ll);
//Removing first element from the list
ll.removeFirst();

        System.out.println("After invoking removeFirst() method: "+ll);
```



```

//Removing first element from the list
ll.removeLast();
    System.out.println("After invoking removeLast() method: "+ll);
//Removing first occurrence of element from the list
ll.removeFirstOccurrence("Gaurav");
    System.out.println("After invoking removeFirstOccurrence() method: "+ll);
//Removing last occurrence of element from the list
ll.removeLastOccurrence("Harsh");
    System.out.println("After invoking removeLastOccurrence() method: "+ll);

//Removing all the elements available in the list
ll.clear();
    System.out.println("After invoking clear() method: "+ll);
}
}

```

Initial list of elements: [Ravi, Vijay, Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking remove(object) method: [Ravi, Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking remove(index) method: [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

Updated list : [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit, Ravi, Hanumat]

After invoking removeAll() method: [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking removeFirst() method: [Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking removeLast() method: [Gaurav, Harsh, Virat, Gaurav, Harsh]

After invoking removeFirstOccurrence() method: [Harsh, Virat, Gaurav, Harsh]

After invoking removeLastOccurrence() method: [Harsh, Virat, Gaurav]

After invoking clear() method: []

### Summary:

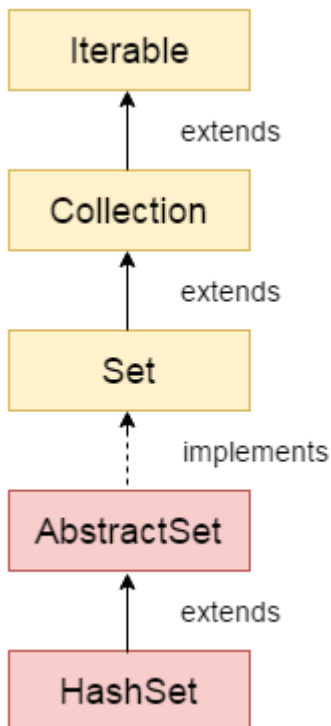
- LinkedList is good for adding elements to the ends, i.e., stacks and queues
- LinkedList performs best while removing objects from middle of the list
- LinkedList implements List, Deque, and Queue interfaces so LinkedList can be used to implement queues and stacks.

### Java HashSet

The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called **hashing**.
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.

- The initial default capacity of HashSet is 16, and the load factor is 0.75.



Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

### Difference between List and Set

A list can contain duplicate elements whereas Set contains unique elements only.

### Hierarchy of HashSet class

The HashSet class extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

### HashSet class declaration

Let's see the declaration for java.util.HashSet class.

1. **public class** HashSet<E> **extends** AbstractSet<E> **implements** Set<E>, Cloneable, Serializable

### Constructors of Java HashSet class

SN	Constructor	Description
1)	HashSet()	It is used to construct a default HashSet.

2)	HashSet(int capacity)	It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.
3)	HashSet(int capacity,	It is used to initialize the capacity of the hash set to the given

	float loadFactor)	integer value capacity and the specified load factor.
4)	HashSet(Collection<? extends E> c)	It is used to initialize the hash set by using the elements of the collection c.

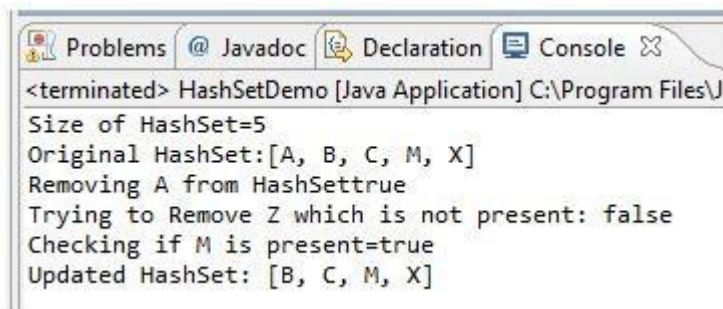
### HashSet Methods

Method	Purpose
public boolean add(Object o)	Adds an object to a HashSet if already not present in HashSet.
public boolean remove(Object o)	Removes an object from a HashSet if found in HashSet.
public boolean contains(Object o)	Returns true if object found else return false
public boolean isEmpty()	Returns true if HashSet is empty else return false
public int size()	Returns number of elements in the HashSet

### Example

```
import java.util.HashSet;
public class HashSetDemo {
    public static void main(String[] args) {
        HashSet<String> hs = new HashSet<String>();
        // Adding element to HashSet
        hs.add("M");
        hs.add("B");
        hs.add("C");
        hs.add("A");
        hs.add("M");
        hs.add("X");
        System.out.println("Size of HashSet=" + hs.size());
        System.out.println("Original HashSet:" + hs);
        System.out.println("Removing A from HashSet: " +
hs.remove("A"));
        System.out.println("Trying to Remove Z which is not present: "
+ hs.remove("Z"));
        System.out.println("Checking if M is present=" +
hs.contains("M"));
        System.out.println("Updated HashSet: " + hs);
    }
}
```

Output:



```
<terminated> HashSetDemo [Java Application] C:\Program Files\J
Size of HashSet=5
Original HashSet:[A, B, C, M, X]
Removing A from HashSettrue
Trying to Remove Z which is not present: false
Checking if M is present=true
Updated HashSet: [B, C, M, X]
```

### Java HashSet Example

Let's see a simple example of HashSet. Notice, the elements iterate in an unordered collection.

```
import java.util.*; class HashSet1{ public
static void main(String args[]){ //Creating
HashSet and adding elements
HashSet<String> set=new HashSet();
set.add("One");      set.add("Two");
set.add("Three");    set.add("Four");
set.add("Five");
    Iterator<String> i=set.iterator();
while(i.hasNext())
{
    System.out.println(i.next());
}
}
```

Five  
One  
Four  
Two  
Three

### Java HashSet example ignoring duplicate elements

In this example, we see that HashSet doesn't allow duplicate elements.

```
import java.util.*; class HashSet2{ public
static void main(String args[]){ //Creating
HashSet and adding elements
HashSet<String> set=new HashSet<String>();
set.add("Ravi");  set.add("Vijay");
set.add("Ravi");  set.add("Ajay");
//Traversing elements
    Iterator<String> itr=set.iterator();
while(itr.hasNext()){
    System.out.println(itr.next());
}
}
```

```
}
```

Ajay  
Vijay  
Ravi

**Java HashSet example to remove elements** Here,  
we see different ways to remove an element.

```
import java.util.*; class
HashSet3{
    public static void main(String args[]){
        HashSet<String> set=new
HashSet<String>();        set.add("Ravi");
set.add("Vijay");        set.add("Arun");
set.add("Sumit");
        System.out.println("An initial list of elements: "+set);
        //Removing specific element from HashSet
set.remove("Ravi");
        System.out.println("After invoking remove(object) method: "+set);
        HashSet<String> set1=new
HashSet<String>();        set1.add("Ajay");
set1.add("Gaurav");        set.addAll(set1);
        System.out.println("Updated List: "+set);
        //Removing all the new elements from HashSet
set.removeAll(set1);
        System.out.println("After invoking removeAll() method: "+set);
        //Removing elements on the basis of specified condition
set.removeIf(str->str.contains("Vijay"));
        System.out.println("After invoking removeIf() method: "+set);
        //Removing all the elements available in the set
set.clear();
        System.out.println("After invoking clear() method: "+set);
    }
}
```

An initial list of elements: [Vijay, Ravi, Arun, Sumit]  
After invoking remove(object) method: [Vijay, Arun, Sumit]  
Updated List: [Vijay, Arun, Gaurav, Sumit, Ajay]  
After invoking removeAll() method: [Vijay, Arun, Sumit]  
After invoking removeIf() method: [Arun, Sumit]  
After invoking clear() method: []

### Java HashSet from another Collection

```
import java.util.*; class
HashSet4{
    public static void main(String args[]){
        ArrayList<String> list=new
ArrayList<String>();      list.add("Ravi");
list.add("Vijay");
        list.add("Ajay");

        HashSet<String> set=new HashSet(list);
set.add("Gaurav");
        Iterator<String> i=set.iterator();
while(i.hasNext())
    {
        System.out.println(i.next());
    }
    }
}
```

Vijay  
Ravi  
Gaurav  
Ajay

### java HashSet Example: Book

Let's see a HashSet example where we are adding books to set and printing all the books.

```

import java.util.*; class
Book {
int id;
String name,author,publisher; int
quantity;
public Book(int id, String name, String author, String publisher, int quantity)
{   this.id = id;   this.name = name;   this.author = author;
this.publisher = publisher;
    this.quantity = quantity;
}
}
public class HashSetExample {
public static void main(String[] args) {
    HashSet<Book> set=new HashSet<Book>();
    //Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw
Hill",4);

    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);

    //Adding Books to HashSet
    set.add(b1);
    set.add(b2);
    set.add(b3);
    //Traversing HashSet
    for(Book b:set){
        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}

```

Output:

```

101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications & Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6

```

### Summary- HashSet

- HashSet provides collection of unique objects
- HashSet is unsorted, unordered and non-indexed based collection class
- HashSet can have only one Null element

### Java LinkedHashSet class

Java LinkedHashSet class is a Hashtable and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

The important points about Java LinkedHashSet class are:

- Java LinkedHashSet class contains unique elements only like HashSet.



- Java LinkedHashSet class provides all optional set operation and permits null elements.
- Java LinkedHashSet class is non synchronized.
- Java LinkedHashSet class maintains insertion order.

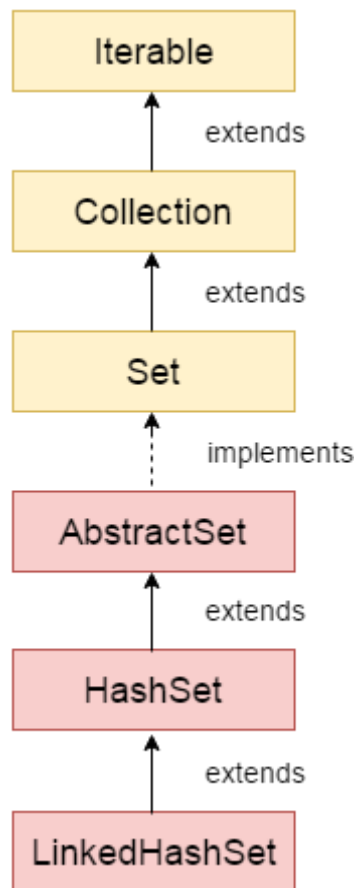
### **Hierarchy of LinkedHashSet class**

The LinkedHashSet class extends HashSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

### **LinkedHashSet class declaration**

Let's see the declaration for java.util.LinkedHashSet class.

**public class** LinkedHashSet<E> **extends** HashSet<E> **implements** Set<E>, Cloneable, Serializable



### Constructors of Java LinkedHashSet class

Constructor	Description
HashSet()	It is used to construct a default HashSet.
HashSet(Collection c)	It is used to initialize the hash set by using the elements of the collection c.
LinkedHashSet(int capacity)	It is used initialize the capacity of the linked hash set to the given integer value capacity.
LinkedHashSet(int capacity, float fillRatio)	It is used to initialize both the capacity and the fill ratio (also called load capacity) of the hash set from its argument.

### LinkedHashSet Methods

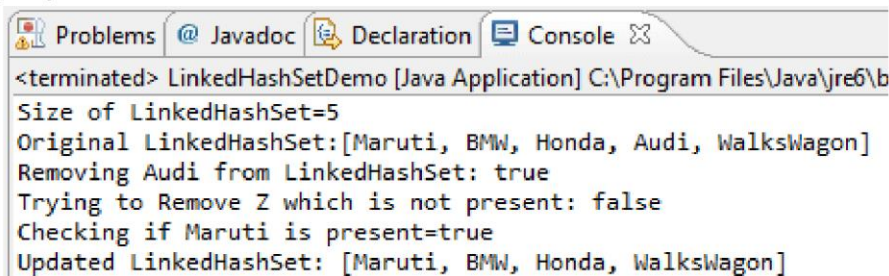
Method	Purpose
public boolean add(Object o)	Adds an object to a LinkedHashSet if already not present in HashSet.

public boolean remove(Object o)	Removes an object from LinkedHashSet if found in HashSet.
public boolean contains(Object o)	Returns true if object found else return false
public boolean isEmpty()	Returns true if LinkedHashSet is empty else return false
public int size()	Returns number of elements in the LinkedHashSet

### Example:

```
import java.util.LinkedHashSet;
public class LinkedHashSetDemo {
    public static void main(String[] args) {
        LinkedHashSet<String> linkedset = new LinkedHashSet<String>();
        // Adding element to LinkedHashSet
        linkedset.add("Maruti");
        linkedset.add("BMW");
        linkedset.add("Honda");
        linkedset.add("Audi");
        linkedset.add("Maruti"); //This will not add new element as Maruti
already exists
        linkedset.add("WalksWagon");
        System.out.println("Size of LinkedHashSet=" + linkedset.size());
        System.out.println("Original LinkedHashSet:" + linkedset);
        System.out.println("Removing Audi from LinkedHashSet: " +
linkedset.remove("Audi"));
        System.out.println("Trying to Remove Z which is not present: "
+ linkedset.remove("Z"));
        System.out.println("Checking if Maruti is present=" +
linkedset.contains("Maruti"));
        System.out.println("Updated LinkedHashSet: " + linkedset);
    }
}
```

### Output:



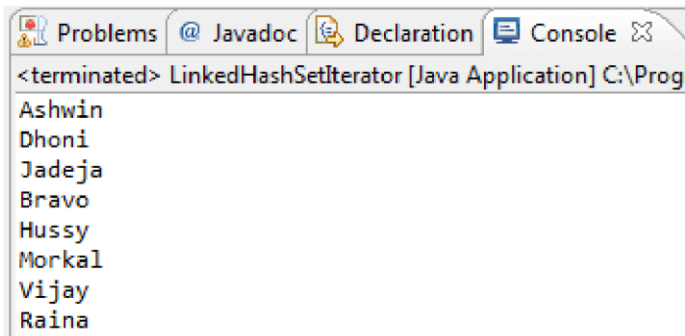
```
<terminated> LinkedHashSetDemo [Java Application] C:\Program Files\Java\jre6\bin\java.exe
Size of LinkedHashSet=5
Original LinkedHashSet:[Maruti, BMW, Honda, Audi, WalksWagon]
Removing Audi from LinkedHashSet: true
Trying to Remove Z which is not present: false
Checking if Maruti is present=true
Updated LinkedHashSet: [Maruti, BMW, Honda, WalksWagon]
```

We can use Iterator object to iterate through our collection. While iterating we can add or remove objects from the collection. Below program demonstrates the use of iterator in LinkedHashSet collection.

#### Java Code:

```
package linkedHashSet;
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.Set;
public class LinkedHashSetIterator {
    public static void main(String[] args) {
        Set<String> myCricketerSet = new LinkedHashSet<String>();
        myCricketerSet.add("Ashwin");
        myCricketerSet.add("Dhoni");
        myCricketerSet.add("Jadeja");
        myCricketerSet.add("Bravo");
        myCricketerSet.add("Hussy");
        myCricketerSet.add("Morkal");
        myCricketerSet.add("Vijay");
        myCricketerSet.add("Raina");
        Iterator<String> setIterator = myCricketerSet.iterator();
        while(setIterator.hasNext()){
            System.out.println(setIterator.next());
        }
    }
}
```

#### Output:



```
<terminated> LinkedHashSetIterator [Java Application] C:\Prog
Ashwin
Dhoni
Jadeja
Bravo
Hussy
Morkal
Vijay
Raina
```

#### Java LinkedHashSet Example

Let's see a simple example of Java LinkedHashSet class. Here you can notice that the elements iterate in insertion order.

```
import java.util.*; class
LinkedHashSet1{ public static void
main(String args[]){ //Creating HashSet
and adding elements
    LinkedHashSet<String> set=new LinkedHashSet();
    set.add("One");          set.add("Two");
    set.add("Three");        set.add("Four");
```

```
        set.add("Five");
        Iterator<String> i=set.iterator();
while(i.hasNext())
    {
        System.out.println(i.next());
    }
}
}
```

One  
Two  
Three  
Four  
Five

### Java LinkedHashSet example ignoring duplicate Elements

```
import java.util.*; class
LinkedHashSet2{
    public static void main(String args[]){
        LinkedHashSet<String> al=new
LinkedHashSet<String>(); al.add("Ravi");
al.add("Vijay"); al.add("Ravi"); al.add("Ajay");
        Iterator<String> itr=al.iterator();
while(itr.hasNext()){
    System.out.println(itr.next());
}
}
}
```

Ravi  
Vijay  
Ajay

### Java LinkedHashSet Example: Book

```
import java.util.*; class
Book {
int id;
String name,author,publisher; int
quantity;
public Book(int id, String name, String author, String publisher, int quantity)
{ this.id = id; this.name = name; this.author = author;
this.publisher = publisher;
this.quantity = quantity;
}
}

public class LinkedHashSetExample {
public static void main(String[] args) {
    LinkedHashSet<Book> hs=new LinkedHashSet<Book>();
    //Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);

    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
    //Adding Books to hash table
    hs.add(b1);
    hs.add(b2);
    hs.add(b3);
    //Traversing hash table
    for(Book b:hs){
        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}
```

#### Output:

```
101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications & Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6
```

#### Summary- LinkedHashSet:

- LinkedHashSet provides collection of unique objects
- LinkedHashSet is unsorted and non-indexed based collection class
- Iteration in LinkedHashSet is as per insertion order

#### Java TreeSet class

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

**The important points about Java TreeSet class are:**

- Java TreeSet class contains unique elements only like HashSet. ○ Java TreeSet class access and retrieval times are quite fast. ○ Java TreeSet class doesn't allow null element.
- Java TreeSet class is non synchronized.
- Java TreeSet class maintains ascending order.

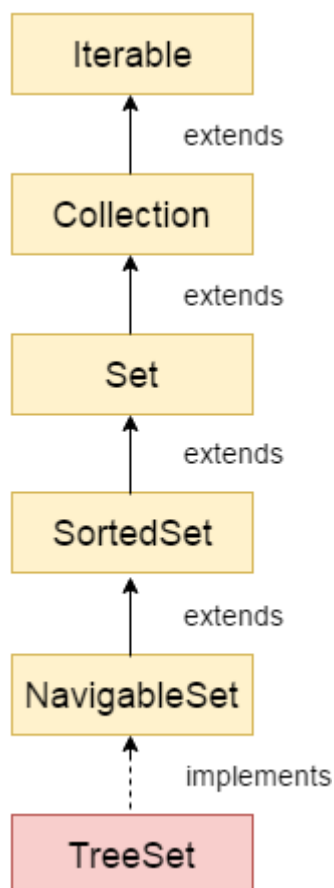
### Hierarchy of TreeSet class

As shown in the above diagram, Java TreeSet class implements the NavigableSet interface. The NavigableSet interface extends SortedSet, Set, Collection and Iterable interfaces in hierarchical order.

### TreeSet class declaration

Let's see the declaration for java.util.TreeSet class.

**public class** TreeSet<E> **extends** AbstractSet<E> **implements** NavigableSet<E>, Cloneable, Serializable



### Constructors of Java TreeSet class

Constructor	Description
TreeSet()	It is used to construct an empty tree set that will be sorted in ascending order according to the natural order of the tree set.
TreeSet(Collection<? extends E> c)	It is used to build a new tree set that contains the elements of the collection c.
TreeSet(Comparator<? super E> comparator)	It is used to construct an empty tree set that will be sorted according to given comparator.
TreeSet(SortedSet<E> s)	It is used to build a TreeSet that contains the elements of the given SortedSet.

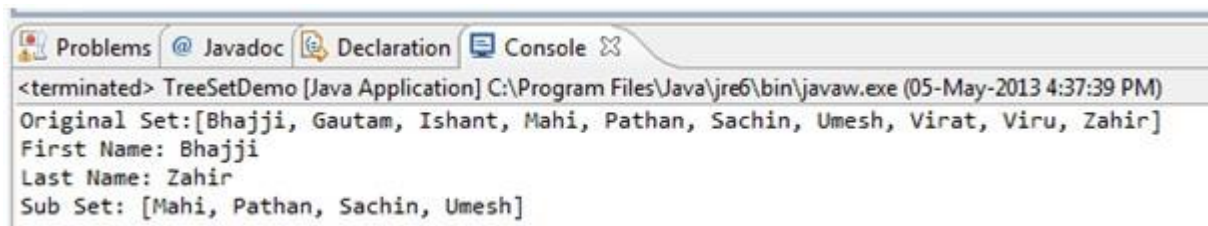


### Important Methods of TreeSet Class

Method	Description
void add(Object o)	Adds the specified element to this set if it is not already present.
void clear()	Removes all of the elements from this set.
Object first()	Returns the first (lowest) element currently in this sorted set.
Object last()	Returns the last (highest) element currently in this sorted set.
boolean isEmpty()	Returns true if this set contains no elements.
boolean remove(Object o)	Removes the specified element from this set if it is present.
SortedSet subSet(Object fromElement, Object toElement)	Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.

```
import java.util.TreeSet;
public class TreeSetDemo {
    public static void main(String[] args) {
        TreeSet<String> playerSet = new TreeSet<String>();
        playerSet.add("Sachin");
        playerSet.add("Zahir");
        playerSet.add("Mahi");
        playerSet.add("Bhaji");
        playerSet.add("Virus");
        playerSet.add("Gautam");
        playerSet.add("Ishant");
        playerSet.add("Umesh");
        playerSet.add("Pathan");
        playerSet.add("Virat");
        playerSet.add("Sachin"); // This is duplicate element so will not be added again
        //below will print list in alphabetic order
        System.out.println("Original Set: " + playerSet);
        System.out.println("First Name: " + playerSet.first());
        System.out.println("Last Name: " + playerSet.last());
        TreeSet<String> newPlySet = (TreeSet<String>) playerSet.subSet("Mahi", "Virat");
        System.out.println("Sub Set: " + newPlySet);
    }
}
```

### Output:



```
<terminated> TreeSetDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (05-May-2013 4:37:39 PM)
Original Set:[Bhajji, Gautam, Ishant, Mahi, Pathan, Sachin, Umesh, Virat, Viru, Zahir]
First Name: Bhajji
Last Name: Zahir
Sub Set: [Mahi, Pathan, Sachin, Umesh]
```

### Java TreeSet Example 1:

Let's see a simple example of Java TreeSet.

```
import java.util.*;
class TreeSet1{
    public static void main(String args[]){
        //Creating and adding elements
        TreeSet<String> al=new TreeSet<String>();
        al.add("Ravi");
        al.add("Vijay");
        al.add("Ravi");
        al.add("Ajay");
        //Traversing elements
        Iterator<String> itr=al.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}
```

### Output:

```
Ajay
Ravi
Vijay
```

### Java TreeSet Example 2:

Let's see an example of traversing elements in descending order.

```

import java.util.*; class
TreeSet2{
    public static void main(String args[]){
        TreeSet<String> set=new TreeSet<String>();
        set.add("Ravi");      set.add("Vijay");
        set.add("Ajay");
        System.out.println("Traversing element through Iterator in descending order");
        Iterator i=set.descendingIterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
    }
}

```

#### Output:

Traversing element through Iterator in descending order  
 Vijay  
 Ravi  
 Ajay  
 Traversing element through NavigableSet in descending order  
 Vijay  
 Ravi  
 Ajay

#### Java TreeSet Example 3:

Let's see an example to retrieve and remove the highest and lowest Value.

```

import java.util.*;
class TreeSet3{
    public static void main(String args[]){
        TreeSet<Integer> set=new TreeSet<Integer>();
        set.add(24);
        set.add(66);
        set.add(12);
        set.add(15);
        System.out.println("Highest Value: "+set.pollFirst());
        System.out.println("Lowest Value: "+set.pollLast());
    }
}

```

#### Output:

Highest Value: 12  
 Lowest Value: 66

#### Java TreeSet Example 4:

In this example, we perform various NavigableSet operations.

```
import java.util.*; class
TreeSet4{
    public static void main(String args[]){
        TreeSet<String> set=new TreeSet<String>();
        set.add("A");      set.add("B");
        set.add("C");      set.add("D");
        set.add("E");
        System.out.println("Initial Set: "+set);

        System.out.println("Reverse Set: "+set.descendingSet());

        System.out.println("Head Set: "+set.headSet("C", true));

        System.out.println("SubSet: "+set.subSet("A", false, "E", true));

        System.out.println("TailSet: "+set.tailSet("C", false));
    }
}
```

Output:

Initial Set: [A, B, C, D, E]

Reverse Set: [E, D, C, B, A]

Head Set: [A, B, C]

SubSet: [B, C, D, E]

TailSet: [D, E]

#### Java TreeSet Example 4:

In this example, we perform various SortedSet operations.

```

import java.util.*;
class TreeSet4{
public static void main(String args[]){
    TreeSet<String> set=new TreeSet<String>();
        set.add("A");
        set.add("B");
        set.add("C");
        set.add("D");
        set.add("E");

        System.out.println("Intial Set: "+set);
        System.out.println("Head Set: "+set.headSet("C"));
        System.out.println("SubSet: "+set.subSet("A", "E"));
        System.out.println("TailSet: "+set.tailSet("C"));
    }
}

```

#### Output:

Intial Set: [A, B, C, D, E]

Head Set: [A, B]

SubSet: [A, B, C, D]

TailSet: [C, D, E]

### Accessing a Java Collection using Iterators

To access elements of a collection, either we can use index if collection is list based or we need to traverse the element. There are three possible ways to traverse through the elements of any collection.

1. Using Iterator interface
2. Using ListIterator interface
3. Using for-each loop

#### 1.Accessing elements using Iterator

Iterator is an interface that is used to iterate the collection elements. It is part of java collection framework. It provides some methods that are used to check and access elements of a collection.

Iterator Interface is used to traverse a list in forward direction, enabling you to remove or modify the elements of the collection. Each collection classes provide `iterator()` method to return an iterator.

#### Iterator Interface Methods

Method	Description
boolean <code>hasNext()</code>	Returns <b>true</b> if there are more elements in the collection. Otherwise, returns false.

E <b>next()</b>	Returns the <b>next element present</b> in the collection. Throws NoSuchElementException if there is not a next element.
void <b>remove()</b>	Removes the current element. Throws IllegalStateException if an attempt is made to call remove() method that is not preceded by a call to <b>next()</b> method.

### Iterator Example

In this example, we are using iterator() method of collection interface that returns an instance of Iterator interface. After that we are using hasNext() method that returns true if collection contains an element and within the loop, obtain each element by calling **next()** method.

```
import java.util.*;
class Demo
{
    public static void main(String[] args)
    {
        ArrayList< String> ar = new ArrayList< String>();
        ar.add("ab");
        ar.add("bc");
        ar.add("cd");
        ar.add("de");
        Iterator it = ar.iterator(); //Declaring Iterator
        while(it.hasNext())
        {
            System.out.print(it.next()+" ");
        }
    }
}
```

ab bc cd de

## 2.Accessing elements using ListIterator

**ListIterator** Interface is used to traverse a list in both **forward** and **backward** direction. It is available to only those collections that implements the **List** Interface.

### Methods of ListIterator:

Method	Description
void <b>add(E obj)</b>	Inserts obj into the list in front of the element that will be returned by the next call to next() method.
boolean <b>hasNext()</b>	Returns true if there is a next element. Otherwise, returns false.
boolean <b>hasPrevious()</b>	Returns true if there is a previous element. Otherwise, returns false.
E <b>next()</b>	Returns the next element. A NoSuchElementException is thrown if there is not a next element.

int <b>nextIndex()</b>	Returns the index of the next element. If there is not a next element, returns the size of the list.
E <b>previous()</b>	Returns the previous element. A NoSuchElementException is thrown if there is not a previous element.
int <b>previousIndex()</b>	Returns the index of the previous element. If there is not a previous element, returns -1.
void <b>remove()</b>	Removes the current element from the list. An IllegalStateException is thrown if remove() method is called before next() or previous() method is invoked.
void <b>set(E obj)</b>	Assigns obj to the current element. This is the element last returned by a call to either next() or previous() method.

### ListIterator Example

Lets create an example to traverse the elements of ArrayList. ListIterator works only with list collection.

```
import java.util.*; class
Demo
{
    public static void main(String[] args)
    {
        ArrayList< String> ar = new ArrayList<
String>();  ar.add("ab");  ar.add("bc");
ar.add("cd");  ar.add("de");
        ListIterator litr = ar.listIterator();
        while(litr.hasNext()) //In forward direction
        {
            System.out.print(litr.next()+" ");
        }
        while(litr.hasPrevious()) //In backward direction
        {
            System.out.print(litr.previous()+" ");
        }
    }
}
```

ab bc cd de de cd bc ab

**3.for-each loop** **for-each** version of **for** loop can also be used for traversing the elements of a collection. But this can only be used if we don't want to modify the contents of a collection and we don't want any **reverse** access. **for-each** loop can cycle through any collection of object that implements **Iterable** interface.

**Exmample:**

```
import java.util.*;
class Demo
{
    public static void main(String[] args)
    {
        LinkedList<String> ls = new LinkedList<String>();
        ls.add("a");
        ls.add("b");
        ls.add("c");
        ls.add("d");
        for(String str : ls)
        {
            System.out.print(str+" ");
        }
    }
}
```

a b c d

**Traversing using for loop**

we can use for loop to traverse the collection elements but only index-based collection can be accessed. For example, list is index-based collection that allows to access its elements using the index value.

```
import java.util.*;
class Demo
{
    public static void main(String[] args)
    {
        LinkedList<String> ls = new
LinkedList<String>();  ls.add("a");
ls.add("b");  ls.add("c");  ls.add("d");
        for(int i = 0; i<ls.size(); i++)
        {
            System.out.print(ls.get(i)+" ");
        }
    }
}
```

a b c d