Collections in Java

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

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

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, LinkedHashSet, 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

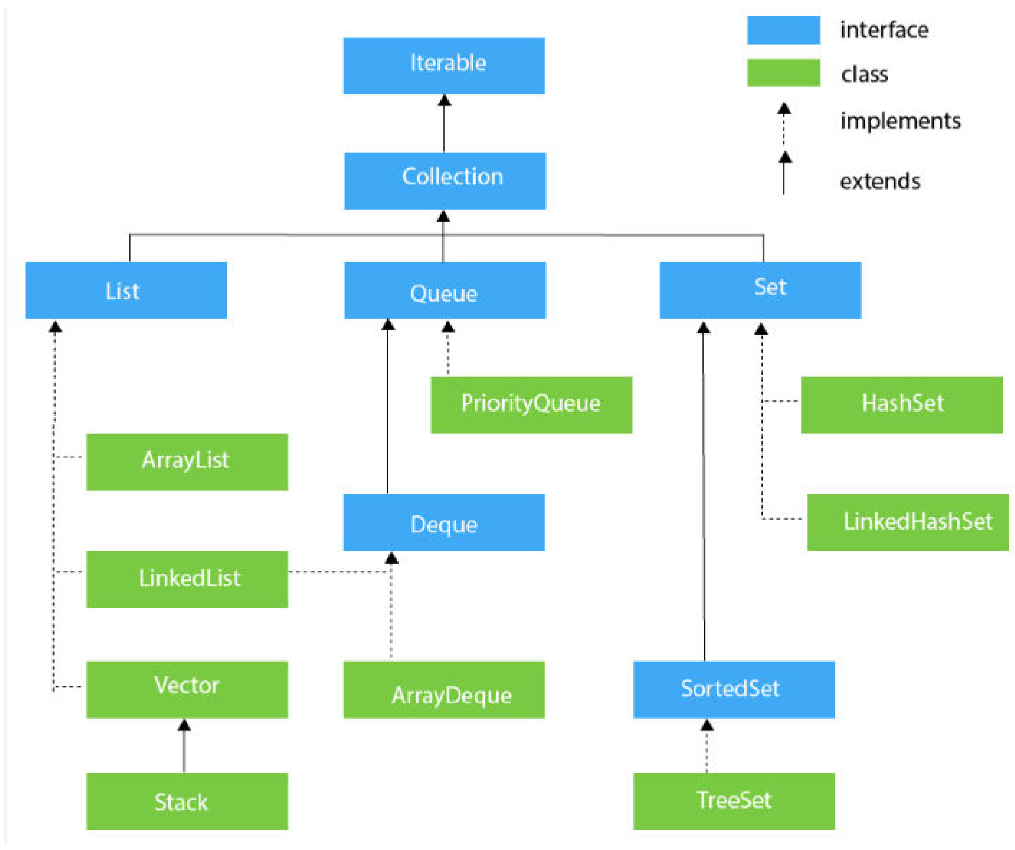
The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

Interfaces and its implementations, i.e., classes

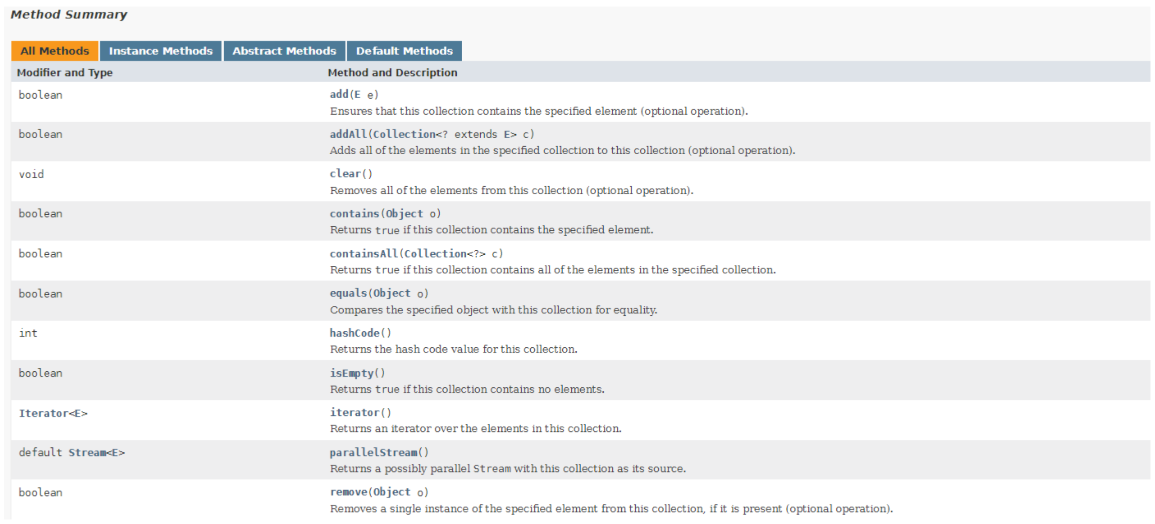
Algorithm

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.



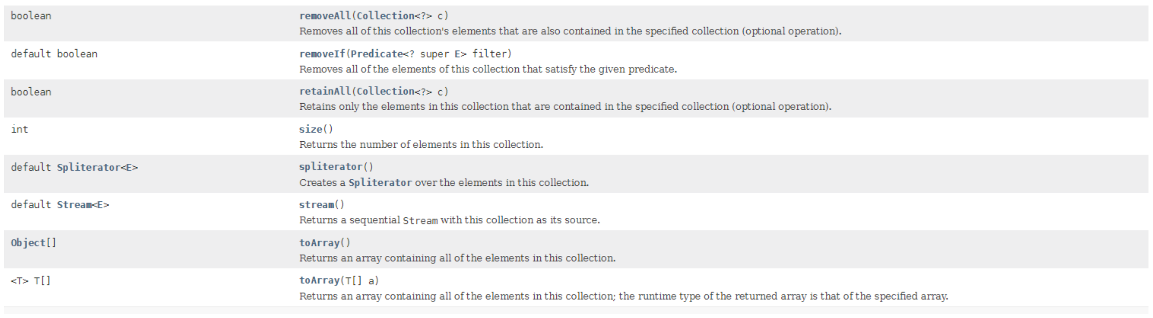
Methods of Collection interface



Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
`boolean`	`add(E e)` Ensures that this collection contains the specified element (optional operation).
`boolean`	`addAll(Collection<? extends E> c)` Adds all of the elements in the specified collection to this collection (optional operation).
`void`	`clear()` Removes all of the elements from this collection (optional operation).
`boolean`	`contains(Object o)` Returns `true` if this collection contains the specified element.
`boolean`	`containsAll(Collection<?> c)` Returns `true` if this collection contains all of the elements in the specified collection.
`boolean`	`equals(Object o)` Compares the specified object with this collection for equality.
`int`	`hashCode()` Returns the hash code value for this collection.
`boolean`	`isEmpty()` Returns `true` if this collection contains no elements.
`Iterator<E>`	`iterator()` Returns an iterator over the elements in this collection.
`default Stream<E>`	`parallelStream()` Returns a possibly parallel `Stream` with this collection as its source.
`boolean`	`remove(Object o)` Removes a single instance of the specified element from this collection, if it is present (optional operation).



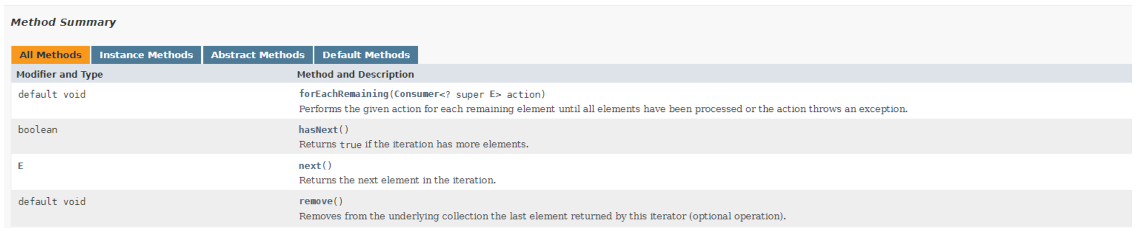
boolean	removeAll(Collection<?> c) Removes all of this collection's elements that are also contained in the specified collection (optional operation).
default boolean	removeIf(Predicate<? super E> filter) Removes all of the elements of this collection that satisfy the given predicate.
boolean	retainAll(Collection<?> c) Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	size() Returns the number of elements in this collection.
default Spliterator<E>	spliterator() Creates a Spliterator over the elements in this collection.
default Stream<E>	stream() Returns a sequential Stream with this collection as its source.
Object[]	toArray() Returns an array containing all of the elements in this collection.
<T> T[]	toArray(T[] a) Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

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:



Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
default void	forEachRemaining(Consumer<? super E> action) Performs the given action for each remaining element until all elements have been processed or the action throws an exception.
boolean	hasNext() Returns true if the iteration has more elements.
E	next() Returns the next element in the iteration.
default void	remove() Removes from the underlying collection the last element returned by this iterator (optional operation).

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.,

Iterator<T> iterator()

It returns the iterator over the elements of type T.

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.

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 :

```
List <data-type> list1= new ArrayList();

List <data-type> list2 = new LinkedList();

List <data-type> list3 = new Vector();

List <data-type> list4 = new Stack();
```

The classes that implement the List interface are given below.

ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

```
import java.util.*;

class JavaExample{

    public static void main(String args[]){

        ArrayList<String> alist=new ArrayList<String>();

        alist.add("Steve");

        alist.add("Tim");

        alist.add("Lucy");

        alist.add("Pat");

        alist.add("Angela");

        alist.add("Tom");


        //displaying elements
```

```
            System.out.println(alist);


            //Adding "Steve" at the fourth position

            alist.add(3, "Steve");


            //displaying elements

            System.out.println(alist);

      }

}
```

LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

```
import java.util.*;

public class JavaExample{

      public static void main(String args[]){


          LinkedList<String> list=new LinkedList<String>();


          //Adding elements to the Linked list

          list.add("Steve");

          list.add("Carl");

          list.add("Raj");


          //Adding an element to the first position
```

```
        list.addFirst("Negan");


        //Adding an element to the last position
        list.addLast("Rick");


        //Adding an element to the 3rd position
        list.add(2, "Glenn");


        //Iterating LinkedList
        Iterator<String> iterator=list.iterator();
        while(iterator.hasNext()){
          System.out.println(iterator.next());
        }
    }
}
```

Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection3{
public static void main(String args[]){
Vector<String> v=new Vector<String>();
v.add("Ayush");
v.add("Amit");
v.add("Ashish");
v.add("Garima");
Iterator<String> itr=v.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Stack

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection4{
public static void main(String args[]){
Stack<String> stack = new Stack<String>();
stack.push("Ayush");
stack.push("Garvit");
stack.push("Amit");
stack.push("Ashish");
stack.push("Garima");
stack.pop();
Iterator<String> itr=stack.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

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:

```
Queue<String> q1 = new PriorityQueue();

Queue<String> q2 = new ArrayDeque();
```

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

PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

```
import java.util.*;

public class TestJavaCollection5{

public static void main(String args[]){

PriorityQueue<String> queue=new PriorityQueue<String>();

queue.add("Amit Sharma");

queue.add("Vijay Raj");

queue.add("JaiShankar");

queue.add("Raj");

System.out.println("head:"+queue.element());

System.out.println("head:"+queue.peek());

System.out.println("iterating the queue elements:");

Iterator itr=queue.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

}
queue.remove();

queue.poll();

System.out.println("after removing two elements:");

Iterator<String> itr2=queue.iterator();

while(itr2.hasNext()){

System.out.println(itr2.next());

}

}

}
```

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:

```
Deque d = new ArrayDeque();
```

ArrayDeque

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

```
import java.util.*;

public class TestJavaCollection6{

public static void main(String[] args) {

//Creating Deque and adding elements

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

deque.add("Gautam");

deque.add("Karan");

deque.add("Ajay");
//Traversing elements

for (String str : deque) {

System.out.println(str);

}

}

}
```

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:

```
Set<data-type> s1 = new HashSet<data-type>();

Set<data-type> s2 = new LinkedHashSet<data-type>();

Set<data-type> s3 = new TreeSet<data-type>();
```

HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example.

```
import java.util.*;

public class TestJavaCollection7{

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());

        }

}

}
```

LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

```
import java.util.*;

public class TestJavaCollection8{

public static void main(String args[]){

LinkedHashSet<String> set=new LinkedHashSet<String>();

set.add("Ravi");

set.add("Vijay");

set.add("Ravi");

set.add("Ajay");

Iterator<String> itr=set.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

}

}

}
```

```
}
```

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:

```
SortedSet<data-type> set = new TreeSet();
```

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

```
import java.util.*;

public class TestJavaCollection9{

public static void main(String args[]){

//Creating and adding elements

TreeSet<String> set=new TreeSet<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());

}

}
```

}

Java ArrayList

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 the List interface here. The ArrayList maintains the insertion order internally.

It inherits the AbstractList class and implements List interface.

The important points about the 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 the array works on an index basis.

In ArrayList, manipulation is a 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.

We can not create an array list of the primitive types, such as int, float, char, etc. It is required to use the required wrapper class in such cases. For example:

ArrayList<int> al = ArrayList<int>(); // does not work

ArrayList<Integer> al = new ArrayList<Integer>(); // works fine

Java ArrayList gets initialized by the size. The size is dynamic in the array list, which varies according to the elements getting added or removed from the list.

ArrayList class declaration

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

public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable

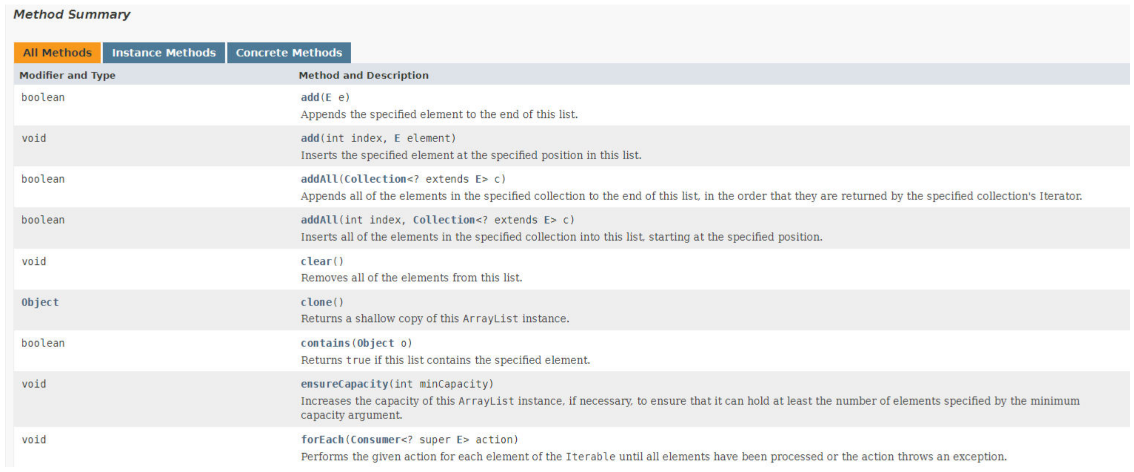
Constructors of ArrayList

Constructor
Description

ArrayList() It is used to build an empty array list.

ArrayList(Collection<? extends E> c) It is used to build an array list that is initialized with the elements of the collection c.

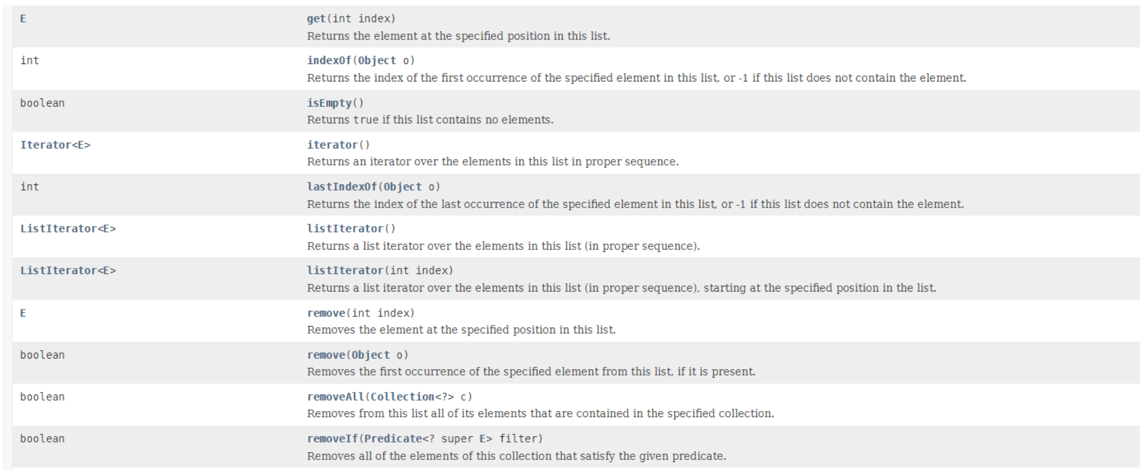
ArrayList(int capacity) It is used to build an array list that has the specified initial capacity.



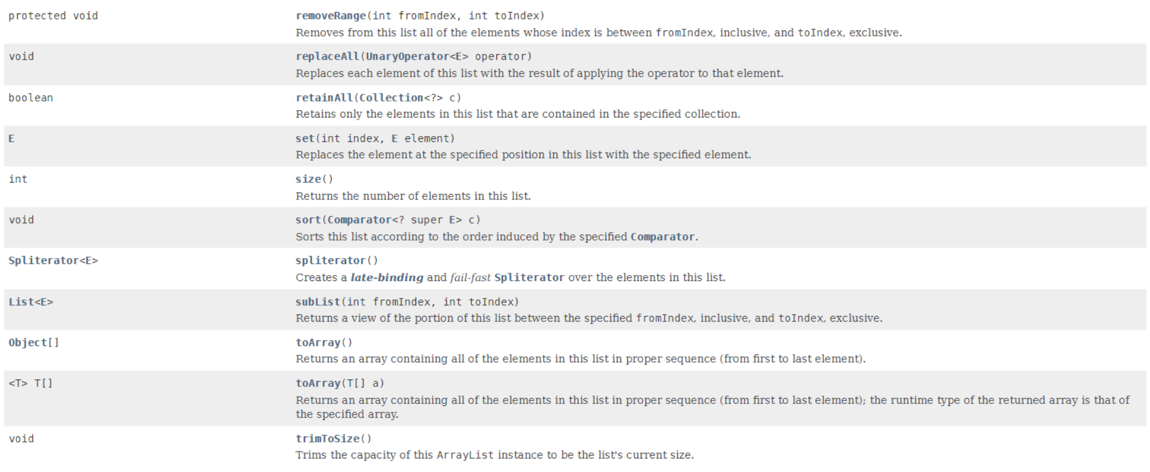
Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
`boolean`	`add(E e)` Appends the specified element to the end of this list.
`void`	`add(int index, E element)` Inserts the specified element at the specified position in this list.
`boolean`	`addAll(Collection<? extends E> c)` Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
`boolean`	`addAll(int index, Collection<? extends E> c)` Inserts all of the elements in the specified collection into this list, starting at the specified position.
`void`	`clear()` Removes all of the elements from this list.
`Object`	`clone()` Returns a shallow copy of this `ArrayList` instance.
`boolean`	`contains(Object o)` Returns `true` if this list contains the specified element.
`void`	`ensureCapacity(int minCapacity)` Increases the capacity of this `ArrayList` instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
`void`	`forEach(Consumer<? super E> action)` Performs the given action for each element of the `Iterable` until all elements have been processed or the action throws an exception.



`E`	`get(int index)` Returns the element at the specified position in this list.
`int`	`indexOf(Object o)` Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`boolean`	`isEmpty()` Returns `true` if this list contains no elements.
`Iterator<E>`	`iterator()` Returns an iterator over the elements in this list in proper sequence.
`int`	`lastIndexOf(Object o)` Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
`ListIterator<E>`	`listIterator()` Returns a list iterator over the elements in this list (in proper sequence).
`ListIterator<E>`	`listIterator(int index)` Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
`E`	`remove(int index)` Removes the element at the specified position in this list.
`boolean`	`remove(Object o)` Removes the first occurrence of the specified element from this list, if it is present.
`boolean`	`removeAll(Collection<?> c)` Removes from this list all of its elements that are contained in the specified collection.
`boolean`	`removeIf(Predicate<? super E> filter)` Removes all of the elements of this collection that satisfy the given predicate.



protected void	removeRange(int fromIndex, int toIndex) Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
void	replaceAll(UnaryOperator<E> operator) Replaces each element of this list with the result of applying the operator to that element.
boolean	retainAll(Collection<?> c) Retains only the elements in this list that are contained in the specified collection.
E	set(int index, E element) Replaces the element at the specified position in this list with the specified element.
int	size() Returns the number of elements in this list.
void	sort(Comparator<? super E> c) Sorts this list according to the order induced by the specified Comparator.
Spliterator<E>	spliterator() Creates a ***late-binding*** and *fail-fast* Spliterator over the elements in this list.
List<E>	subList(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
Object[]	toArray() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	toArray(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.
void	trimToSize() Trims the capacity of this ArrayList instance to be the list's current size.

Java Non-generic Vs. Generic Collection

Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.

Java new generic collection allows you to have only one type of object in a collection. Now it is type-safe, so typecasting is not required at runtime.

Let's see the old non-generic example of creating a Java collection.

ArrayList list=new ArrayList();//creating old non-generic arraylist

Let's see the new generic example of creating java collection.

ArrayList<String> list=new ArrayList<String>();//creating new generic arraylist

In a generic collection, we specify the type in angular braces. Now ArrayList is forced to have the only specified type of object in it. If you try to add another type of object, it gives a compile-time error.

Java ArrayList Example

```
import java.util.*;

class Main{

    public static void main(String args[]){

        ArrayList<String> city_List=new ArrayList<String>();


        city_List.add("Delhi");

        city_List.add("Mumbai");
```

```
            city_List.add("Chennai");

            city_List.add("Kolkata");

            System.out.println("Initial ArrayList:" + city_List);

            city_List.add(1, "NYC");

             System.out.println("\nrrayList after adding element at index 1:" + city_List);

            ArrayList<String> more_Cities = new ArrayList<String>(Arrays.asList("Pune", "Hyderabad"));

                        city_List.addAll(4,more_Cities);

                        System.out.println("\nArrayList after adding list at index 4:" + city_List);

      }

}
```

ArrayList Add To The Front

```
import java.util.ArrayList;

  public class Main {

        public static void main(String[] args) {

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

                numList.add(5);

                numList.add(7);

                numList.add(9);

              System.out.println("Initial ArrayList:");

                System.out.println(numList);




                numList.add(0, 3);

                numList.add(0, 1);


                System.out.println("ArrayList after adding elements at the beginning:");

                System.out.println(numList);

        }
}
```

ArrayList remove:-

```
import java.util.*;
class Main{
    public static void main(String args[]){
        ArrayList<String> city_List=new ArrayList<String(Arrays.asList("Delhi","Mumbai","Chennai",
                "Kolkata", "Pune", "Hyderabad"));
                 System.out.println("Initial ArrayList:" + city_List);

        city_List.remove(2);
        System.out.println("\nArrayList after removing element at index 2:" + city_List);

        city_List.remove("Kolkata");
        System.out.println("\nArrayList after removing element -> Kolkata:" + city_List);

        ArrayList<String> newCities=new ArrayList<String>(Arrays.asList("Delhi","Hyderabad"));
        city_List.removeAll(newCities);
        System.out.println("\nArrayList after call to removeAll:" + city_List);
    }
}
```

ArrayList size (Length):-

```
import java.util.ArrayList;
public class Main
{
        public static void main(String [] args)
        {

                ArrayList<Integer> evenList=new ArrayList<Integer>(5);
                System.out.println("Initial size: "+evenList.size());
                evenList.add(2);
                evenList.add(4);
                evenList.add(6);
                evenList.add(8);
                evenList.add(10);
                System.out.println("Original List: " + evenList);
                System.out.println("ArrayList Size after add operation: "+evenList.size());

                evenList.ensureCapacity(10);
                evenList.add(12);
                evenList.add(14);
                System.out.println("ArrayList Size after ensureCapacity() call and add operation: 
"+evenList.size());
                evenList.trimToSize();
                System.out.println("ArrayList Size after trimToSize() operation: "+evenList.size());
                System.out.println("ArrayList final: ");
                for(int num: evenList){
                        System.out.print(num + " ");
                }
            }
}
```

ArrayList contains:-

```
import java.util.ArrayList;
public class Main {
    public static void main(String[] args) {

        ArrayList<String> colorsList = new ArrayList<String>();
        colorsList.add("Red");
        colorsList.add("Green");
        colorsList.add("Blue");
        colorsList.add("White");

        System.out.println("ArrayList contains ('Red Green'): "
                                                +colorsList.contains("Red Green"));
        System.out.println("ArrayList contains ('Blue'): "
                                                  +colorsList.contains("Blue"));
        System.out.println("ArrayList contains ('Yellow'): "
                                               +colorsList.contains("Yellow"));
        System.out.println("ArrayList contains ('White'): "
                                                +colorsList.contains("White"));
    }
}
```

ArrayList get:-

```
import java.util.ArrayList;
public class Main {
    public static void main(String[] args) {
        //create and initialize colorsList
        ArrayList<String> colorsList = new ArrayList<String>();

        colorsList.add("Red");

        colorsList.add("Green");

        colorsList.add("Blue");

        colorsList.add("White");

        //call get () method to retrieve value at index 2

        System.out.println("Entry at index 2 before call to set: " + colorsList.get(2));


        //replace the value at index 2 with new value

        colorsList.set(2,"Yellow");


        //print the value at index 2 again

        System.out.println("Entry at index 2 after call to set: " + colorsList.get(2));

    }

}
```

ArrayList clear:-

ArrayList isEmpty:-

```
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {


        ArrayList<String> colorsList = new ArrayList<String>();

        colorsList.add("Red");

        colorsList.add("Green");

        colorsList.add("Blue");
        colorsList.add("White");

        System.out.println("The ArrayList: " + colorsList);

        colorsList.clear();

        System.out.println("Is ArrayList empty after clear ()? :" + colorsList.isEmpty());

    }

}
```

ArrayList indexOf:-

ArrayList lastIndexOf:-

```
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {


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

        intList.add(1);

        intList.add(1);

        intList.add(2);

        intList.add(3);

        intList.add(5);

        intList.add(3);

        intList.add(2);

        intList.add(1);

        intList.add(1);


        System.out.println("The ArrayList: " + intList);
```

```
            System.out.println("indexOf(1) : " + intList.indexOf(1));
            System.out.println("lastIndexOf(1) : " + intList.lastIndexOf(1));
            System.out.println("indexOf(2) : " + intList.indexOf(2));
            System.out.println("lastIndexOf(2) : " + intList.lastIndexOf(2));
             System.out.println("indexOf(3) : " + intList.indexOf(3));
            System.out.println("lastIndexOf(3) : " + intList.lastIndexOf(3));
             System.out.println("indexOf(5) : " + intList.indexOf(5));
            System.out.println("lastIndexOf(5) : " + intList.lastIndexOf(5));
      }
}
```

ArrayList to Array:-

```
import java.util.*;
public class Main {
        public static void main(String[] args) {
                // define and initialize ArrayList
                ArrayList<Integer> intList = new ArrayList<Integer>();
                intList.add(10);
                intList.add(20);
                intList.add(30);
                intList.add(40);
                intList.add(50);
            System.out.println("ArrayList: "    + intList);
              Integer myArray[] = new Integer[intList.size()];
                myArray = intList.toArray(myArray);
                System.out.println("Array from ArrayList:" + Arrays.toString(myArray));
        }
}
```

ArrayList clone:-

```
import java.util.ArrayList;
public class Main {
    public static void main(String a[]){
        ArrayList<String> fruitsList = new ArrayList<String>();

        fruitsList.add("Apple");
        fruitsList.add("Orange");
        fruitsList.add("Melon");
        fruitsList.add("Grapes");
        System.out.println("Original ArrayList: "+fruitsList);

        ArrayList<String> clone_list = (ArrayList<String>)fruitsList.clone();
        System.out.println("Cloned ArrayList: "+ clone_list);

        fruitsList.add("Mango");
        fruitsList.remove("Orange");

        System.out.println("\nOriginal ArrayList after add & remove:"+fruitsList);
        System.out.println("Cloned ArrayList after original changed:"+clone_list);
    }
}
```

ArrayList subList:-

```
import java.util.ArrayList;

import java.util.List;

class Main{

  public static void main(String a[]){

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

          intList.add(5);

          intList.add(10);

          intList.add(15);

          intList.add(20);

          intList.add(25);

          intList.add(30);

          intList.add(35);

          intList.add(40);

          intList.add(45);

          intList.add(50);


          System.out.println("Original ArrayList: "+intList);


          ArrayList<Integer> sub_ArrayList = new ArrayList<Integer>(intList.subList(2, 6));

          System.out.println("Sublist of given ArrayList: "+sub_ArrayList);

    }

}
```

ArrayList retainAll:-

```
import java.util.*;
```

```
class Main{
  public static void main(String args[]){
    ArrayList<String> colorsList=new ArrayList<String>();
    colorsList.add("Red");
    colorsList.add("Green");
    colorsList.add("Blue");
    colorsList.add("Yellow");
    System.out.println("Original ArrayList:" + colorsList);

    ArrayList<String> color_collection=new ArrayList<String>();
    color_collection.add("Red");
    color_collection.add("Blue");
    System.out.println("Collection elements to be retained in the list:" + color_collection);

    colorsList.retainAll(color_collection);
    System.out.println("ArrayList after retainAll call:" + colorsList);
  }
}
```

ArrayList Iterator:-

```
import java.util.*;
class Main{
  public static void main(String args[]){
                ArrayList<String> cities=new ArrayList<String>();
                cities.add("Mumbai");
                cities.add("Pune");
                cities.add("Hyderabad");
                cities.add("Delhi");

                System.out.println("List contents using Iterator () method:");
                Iterator iter=cities.iterator();
                while(iter.hasNext()){
                        System.out.print(iter.next() + " ");
                }

                System.out.println("\n\nList contents using listIterator () method:");
                ListIterator<String> list_iter=cities.listIterator();
                while(list_iter.hasNext())    {
                    System.out.print(list_iter.next() + " ");
                }
    }
}
```

Add Array To ArrayList In Java:-

```
import java.util.*;
class Main{
      public static void main(String args[]){
            ArrayList<String> city_List=new ArrayList<String>();
            city_List.add("Delhi");
            city_List.add("Mumbai");
            city_List.add("Chennai");
            city_List.add("Kolkata");
            System.out.println("\nInitial ArrayList :" + city_List);


            String[] myArray = new String[]{"Cochin", "Goa"};
            Collections.addAll(city_List,myArray);


            System.out.println("\nArrayList after adding array :" + city_List);
      }
}
```

Sort ArrayList In Java:-

```
import java.util.*;
public class Main    {
        public static void main(String args[]){
              ArrayList<String> colorsList = new ArrayList<String>();
              colorsList.add("Red");
              colorsList.add("Green");
              colorsList.add("Blue");
              colorsList.add("Yellow");


              System.out.println("Initial ArrayList:" + colorsList);


              Collections.sort(colorsList);


              System.out.println("\nArrayList sorted in ascending order:");
              System.out.println(colorsList);


                Collections.sort(colorsList, Collections.reverseOrder());
        System.out.println("\nArrayList sorted in descending order:");
        System.out.println(colorsList);
    }
}
```

Reverse An ArrayList In Java:-

```
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args)
    {
        ArrayList<Integer> oddList = new ArrayList<Integer>();
        oddList.add(1);
        oddList.add(3);
        oddList.add(5);
        oddList.add(7);
        oddList.add(9);
        System.out.print("Initial ArrayList: " + oddList);

        Collections.reverse(oddList);
        System.out.print("\nReversed ArrayList: " + oddList);
   }
}
```

Remove Duplicates From An ArrayList In Java:-

```
import java.util.*;
import java.util.stream.Collectors;
```

```
public class Main {

        public static void main(String[] args)    {

                ArrayList<Integer> numList = new ArrayList<>

                (Arrays.asList(1, 2, 3, 1, 3, 5, 5, 6, 6, 7, 7, 8, 8));


                System.out.println("Original ArrayList:" + numList);


                //Use Java 8 stream().distinct()    method to remove duplicates from the list

                List<Integer> distinctList = numList.stream().distinct().collect(Collectors.toList());

                //print the new list

                System.out.println("ArrayList without duplicates:" + distinctList);

        }

}
```

Shuffle (Randomize) An ArrayList In Java:-

```
import java.util.*;


public class Main {

        public static void main(String[] args)

        {

                //create and initialize a String ArrayList

                ArrayList<String> strlist = new ArrayList<String>();
                strlist.add("east");

                strlist.add("west");

                strlist.add("north");

                strlist.add("south");

                strlist.add("southwest");

                strlist.add("northeast");

                //print the original list

                System.out.println("Original ArrayList : \n" + strlist);


                //shuffle the ArrayList without random function

                Collections.shuffle(strlist);

                System.out.println("\nShuffled ArrayList without Random() : \n"

                                    + strlist);


                // shuffle the ArrayList with random() function

                Collections.shuffle(strlist, new Random());

                System.out.println("\nShuffled ArrayList with Random() : \n" + strlist);


                // use random (2) to shuffle the ArrayList

                Collections.shuffle(strlist, new Random(2));

                System.out.println("\nShuffled ArrayList with Random(2) : \n" + strlist);

        }

}
```

User-defined class objects in Java ArrayList

```
class Student{
    int rollno;

    String name;

    int age;

    Student(int rollno,String name,int age){

      this.rollno=rollno;

      this.name=name;

      this.age=age;

    }

}


import java.util.*;

  class User

{

  public static void main(String args[]){



    Student s1=new Student(1,"Ajay",21);

    Student s2=new Student(2,"Manish",30);

    Student s2=new Student(3,"Sandeep",29);

        ArrayList<Student> al=new ArrayList<Student>();

    al.add(s1);

    al.add(s2);

    al.add(s3);

        Iterator itr=al.iterator();


    while(itr.hasNext()){
        Student st=(Student)itr.next();

        System.out.println(st.rollno+" "+st.name+" "+st.age);

    }

  }

}
```

Java LinkedList class

Java LinkedList class hierarchy

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.

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.

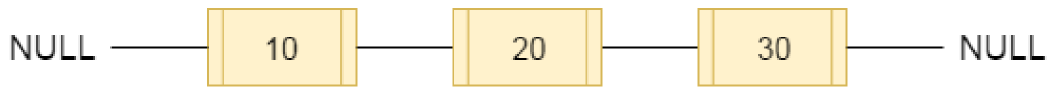
Java LinkedList class can be used as a list, stack or queue.

Hierarchy of LinkedList class

As shown in the above diagram, Java LinkedList class extends AbstractSequentialList class and implements List and Deque interfaces.

Doubly Linked List

In the case of a doubly linked list, we can add or remove elements from both sides.



LinkedList class declaration

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

```
public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable
```

Constructors of Java LinkedList

Linked List Implementation In Java

```
import java.util.*;

public class LinkedList1{

  public static void main(String args[]){



    LinkedList<String> al=new LinkedList<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());

    }

  }

}
```

Java LinkedList example to add elements

```
import java.util.*;

public class JavaExample{

      public static void main(String args[]){



          LinkedList<String> list=new LinkedList<String>();
          //Adding elements to the Linked list

          list.add("Steve");

          list.add("Carl");

          list.add("Raj");


          //Adding an element to the first position

          list.addFirst("Negan");


          //Adding an element to the last position

          list.addLast("Rick");


          //Adding an element to the 3rd position

          list.add(2, "Glenn");


          //Iterating LinkedList

          Iterator<String> iterator=list.iterator();

          while(iterator.hasNext()){

              System.out.println(iterator.next());

          }

      }

}


import java.util.*;

public class LinkedList2{
  public static void main(String args[]){
  LinkedList<String> ll=new LinkedList<String>();
                    System.out.println("Initial list of elements: "+ll);
                    ll.add("Ravi");
                    ll.add("Vijay");
                    ll.add("Ajay");
                    System.out.println("After invoking add(E e) method: "+ll);
                    //Adding an element at the specific position
                    ll.add(1, "Gaurav");
                    System.out.println("After invoking add(int index, E element) method: "+ll);
                    LinkedList<String> ll2=new LinkedList<String>();
                    ll2.add("Sonoo");
                    ll2.add("Hanumat");
                    //Adding second list elements to the first list
                    ll.addAll(ll2);
                    System.out.println("After invoking addAll(Collection<? extends E> c) method: "+ll);
                    LinkedList<String> ll3=new LinkedList<String>();
                    ll3.add("John");
                    ll3.add("Rahul");
                    //Adding second list elements to the first list at specific position
                    ll.addAll(1, ll3);
                    System.out.println("After invoking addAll(int index, Collection<? extends E> c)
method:"+ll);
                    //Adding an element at the first position
                    ll.addFirst("Lokesh");
                    System.out.println("After invoking addFirst(E e) method: "+ll);
```

```
                //Adding an element at the last position
                ll.addLast("Harsh");
                System.out.println("After invoking addLast(E e) method: "+ll);

  }
}
```

Java example of removing elements from the LinkedList

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

        LinkedList<String> list=new LinkedList<String>();

        //Adding elements to the Linked list
        list.add("Steve");
        list.add("Carl");
        list.add("Raj");
        list.add("Negan");
        list.add("Rick");

        //Removing First element
        //Same as list.remove(0);
        list.removeFirst();
        //Removing Last element
        list.removeLast();

        //Iterating LinkedList
        Iterator<String> iterator=list.iterator();
        while(iterator.hasNext()){
            System.out.print(iterator.next()+" ");
        }

        //removing 2nd element, index starts with 0
        list.remove(1);

        System.out.print("\nAfter removing second element: ");
        //Iterating LinkedList again
        Iterator<String> iterator2=list.iterator();
        while(iterator2.hasNext()){
            System.out.print(iterator2.next()+" ");
        }
    }
}
```

Java LinkedList example to remove elements

```
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);

      }

   }
```

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());
                  }

  }
}
```

Java LinkedList 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 LinkedListExample {

public static void main(String[] args)

  {

        List<Book> list=new LinkedList<Book>();

        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);


        list.add(b1);

        list.add(b2);

        list.add(b3);


        for(Book b:list){

        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);

        }

}

}
```

Get first and last elements from LinkedList example:-

Here we have a LinkedList of String type and we are getting first and last element from it using getFirst() and getLast() methods of LinkedList class. Method definition and description are as follows:

1) public E getFirst(): Returns the first element in this list.

2) public E getLast(): Returns the last element in this list.

```
}

import java.util.LinkedList;

public class GetFirstAndLast {


  public static void main(String[] args) {


        // Create a LinkedList

        LinkedList<String> linkedlist = new LinkedList<String>();


        // Add elements to LinkedList

        linkedlist.add("Item1");

        linkedlist.add("Item2");

        linkedlist.add("Item3");

        linkedlist.add("Item4");

        linkedlist.add("Item5");

        linkedlist.add("Item6");


        // Getting First element of the List

        Object firstElement = linkedlist.getFirst();

        System.out.println("First Element is: "+firstElement);


        // Getting Last element of the List

        Object lastElement = linkedlist.getLast();

        System.out.println("Last Element is: "+lastElement);

  }

}
```

Get element from specific index of LinkedList example

In this example we are gonna see how to get an element from specific index of LinkedList using get(int index) method:

public E get(int index): Returns the element at the specified position in this list.

```
import java.util.LinkedList;


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


        // Creating LinkedList of String Elements
        LinkedList<String> linkedlist = new LinkedList<String>();


        // Populating it with String values
        linkedlist.add("AA");
        linkedlist.add("BB");
        linkedlist.add("CC");
        linkedlist.add("DD");
        linkedlist.add("EE");


        System.out.println("LinkedList Elements : ");
        //get(i) returns element present at index i
        for(int i=0; i < linkedlist.size(); i++){
            System.out.println("Element at index "+i+" is: "+linkedlist.get(i));
        }
  }
}
```

Search elements in LinkedList example:-

public int indexOf(Object o): Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

public int lastIndexOf(Object o): Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.

Example

Here we are having a LinkedList of String elements and we are searching a string using indexOf() and lastIndexOf() methods of LinkedList class.

```
import java.util.LinkedList;
public class SearchInLinkedList {

    public static void main(String[] args) {

          // Step1: Create a LinkedList
          LinkedList<String> linkedlist = new LinkedList<String>();

          // Step2: Add elements to LinkedList
          linkedlist.add("Tim");
          linkedlist.add("Rock");
          linkedlist.add("Hulk");
          linkedlist.add("Rock");
          linkedlist.add("James");
          linkedlist.add("Rock");

          //Searching first occurrence of element
          int firstIndex = linkedlist.indexOf("Rock");
          System.out.println("First Occurrence: " + firstIndex);

          //Searching last occurrence of element
          int lastIndex = linkedlist.lastIndexOf("Rock");
          System.out.println("Last Occurrence: " + lastIndex);
    }
}
```

Get sub List from LinkedList example

```
import java.util.LinkedList;
import java.util.Iterator;
import java.util.List;
public class SublistExample {

  public static void main(String[] args) {

        // Create a LinkedList
        LinkedList<String> linkedlist = new LinkedList<String>();
        }


        // Add elements to LinkedList

        linkedlist.add("Item1");

        linkedlist.add("Item2");

        linkedlist.add("Item3");

        linkedlist.add("Item4");

        linkedlist.add("Item5");

        linkedlist.add("Item6");

        linkedlist.add("Item7");


        // Displaying LinkedList elements

        System.out.println("LinkedList elements:");

        Iterator it= linkedlist.iterator();

        while(it.hasNext()){

              System.out.println(it.next());

        }


        // Obtaining Sublist from the LinkedList

        List sublist = linkedlist.subList(2,5);


        // Displaying SubList elements

        System.out.println("\nSub List elements:");

        Iterator subit= sublist.iterator();

        while(subit.hasNext()){

              System.out.println(subit.next());

        }
```

```
        /* Any changes made to Sub List will be reflected
          * in the original List. Lets take this example - We
          * are removing element "Item4" from sublist and it
          * should be removed from original list too. Observe
          * the Output of this part of the program.
          */
        sublist.remove("Item4");
        System.out.println("\nLinkedList elements After remove:");
        Iterator it2= linkedlist.iterator();
        while(it2.hasNext()){
              System.out.println(it2.next());
        }
  }
}
```

Java – LinkedList Iterator example:-

Example

The steps we followed in the below program are:

1) Create a LinkedList

2) Add element to it using add(Element E) method

3) Obtain the iterator by calling iterator() method

4) Traverse the list using hasNext() and next() method of Iterator class.

```
}

import java.util.LinkedList;
import java.util.Iterator;
public class IteratorExample {

  public static void main(String[] args) {

        // Create a LinkedList
        LinkedList<String> linkedlist = new LinkedList<String>();

        // Add elements to LinkedList
        linkedlist.add("Delhi");
        linkedlist.add("Agra");
        linkedlist.add("Mysore");
        linkedlist.add("Chennai");
        linkedlist.add("Pune");

        // Obtaining Iterator
        Iterator it = linkedlist.iterator();

        // Iterating the list in forward direction
        System.out.println("LinkedList elements:");
        while(it.hasNext()){
              System.out.println(it.next());
        }
  }
```

```
}
```

LinkedList ListIterator example

```
import java.util.LinkedList;
import java.util.ListIterator;
public class ListIteratorExample {

  public static void main(String[] args) {

        // Create a LinkedList
        LinkedList<String> linkedlist = new LinkedList<String>();

        // Add elements to LinkedList
        linkedlist.add("Delhi");
        linkedlist.add("Agra");
        linkedlist.add("Mysore");
        linkedlist.add("Chennai");
        linkedlist.add("Pune");

        // Obtaining ListIterator
        ListIterator listIt = linkedlist.listIterator();

        // Iterating the list in forward direction
        System.out.println("Forward iteration:");
        while(listIt.hasNext()){
            System.out.println(listIt.next());
        }


        // Iterating the list in backward direction
        System.out.println("\nBackward iteration:");
        while(listIt.hasPrevious()){
            System.out.println(listIt.previous());
        }
  }
}
```

Iterate a LinkedList in reverse sequential order

```
import java.util.LinkedList;
import java.util.Iterator;
class LinkedListDemo {

    public static void main(String[] args) {

        // create a LinkedList
        LinkedList<String> list = new LinkedList<String>();

        // Adding elements to the LinkedList
        list.add("Element1");
        list.add("Element2");
        list.add("Element3");
        list.add("Element4");
```

```
          // Displaying LinkedList elements
          System.out.println("LinkedList elements: "+list);

          /* public Iterator<E> descendingIterator(): Returns an
            * iterator over the elements in this list in reverse
            * sequential order. The elements will be returned in
            * order from last (tail) to first (head).
            */
          Iterator it = list.descendingIterator();

          // Displaying list in reverse order
          System.out.println("Elements in Reverse Order:");
          while (it.hasNext()) {
                System.out.println(it.next());
          }
    }
}
```

Java – Replace element in a LinkedList example

```
import java.util.LinkedList;
public class ReplaceInLinkedList {

  public static void main(String[] args) {

        // Create a LinkedList
}

        LinkedList<String> linkedlist = new LinkedList<String>();

        // Add elements to LinkedList
        linkedlist.add("Cobol");
        linkedlist.add("JCL");
        linkedlist.add("C++");
        linkedlist.add("C#");
        linkedlist.add("Java");

        // Displaying Elements before replace
        System.out.println("Before Replace:");
        for(String str: linkedlist){
            System.out.println(str);
        }

        // Replacing 3rd Element with new value
        linkedlist.set(2, "NEW VALUE");
        System.out.println("\n3rd Element Replaced \n");

        // Displaying Elements after replace
        System.out.println("After Replace:");
        for(String str2: linkedlist){
            System.out.println(str2);
        }
    }
```

```
}
```

Java – Check if a particular element exists in LinkedList example

```
import java.util.LinkedList;

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

        // Creating LinkedList of String Elements
        LinkedList<String> linkedlist = new LinkedList<String>();

        // Populating it with String values
        linkedlist.add("AA");
        linkedlist.add("BB");
        linkedlist.add("CC");
        linkedlist.add("DD");
        linkedlist.add("EE");

        // contains() method checks whether the element exists
        if (linkedlist.contains("CC")) {
              System.out.println("Element CC is present in List");
        } else {
              System.out.println("List doesn't have element CC");
          }
        //Checking for element FF
        if (linkedlist.contains("FF")) {
            System.out.println("Element FF is present in List");
        } else {
              System.out.println("List doesn't have element FF");
          }
  }
}
```

Java – Get the index of last occurrence of an element in LinkedList

```
import java.util.LinkedList;
class LinkedListExample {

    public static void main(String[] args) {

          // create a LinkedList
          LinkedList<String> list = new LinkedList<String>();

          // Add elements
          list.add("AA");
          list.add("BB");
          list.add("CC");
          list.add("AA");
          list.add("DD");
          list.add("AA");
```

```
        list.add("EE");

        // Display LinkedList elements
        System.out.println("LinkedList elements: "+list);

        // get the index of last occurrence of element "AA"
        /* public int lastIndexOf(Object o): Returns the index
         * of the last occurrence of the specified element in
         * this list, or -1 if this list does not contain the
         * element.
         */
        System.out.println("LastIndex of AA:"+list.lastIndexOf("AA"));

        // get the index of last occurrence of element "ZZ"
        /* Note: The element ZZ does not exist in the list so
         * the method lastIndexOf would return -1 for it.
         */
        System.out.println("LastIndex of ZZ:"+list.lastIndexOf("ZZ"));
    }
}
```

LinkedList push() and pop() methods – Java:-

```
import java.util.LinkedList;
class LinkedListExample {
    public static void main(String[] args) {

        // Create a LinkedList of Strings
        LinkedList<String> list = new LinkedList<String>();

        // Add few Elements
        list.add("Jack");
        list.add("Robert");
        list.add("Chaitanya");
        list.add("kate");

        // Display LinkList elements
        System.out.println("LinkedList contains: "+list);

        // push Element the list
        list.push("NEW ELEMENT");

        // Display after push operation
        System.out.println("LinkedList contains: "+list);
    }
}
```

LinkedList.pop()

```
import java.util.LinkedList;
class LinkedListPopDemo{

   public static void main(String[] args) {

       // Create a LinkedList of Strings
       LinkedList<String> list = new LinkedList<String>();

       // Add few Elements
       list.add("Jack");
       list.add("Robert");
       list.add("Chaitanya");
       list.add("kate");

       // Display LinkList elements
       System.out.println("LinkedList before: "+list);

       // pop Element from list and display it
       System.out.println("Element removed: "+list.pop());

       // Display after pop operation
       System.out.println("LinkedList after: "+list);
   }
}
```

Java – LinkedList poll(), pollFirst() and pollLast() methods

LinkedList.poll()

Retrieves and removes the head (first element) of this list.

```
import java.util.LinkedList;

class LinkedListPollMethod{

  public static void main(String[] args) {

     // Create a LinkedList of Strings

     LinkedList<String> list = new LinkedList<String>();

     // Add few Elements

     list.add("Element1");

     list.add("Element2");

     list.add("Element3");

     list.add("Element4");

     // Display LinkList elements

     System.out.println("LinkedList before: "+list);

     /* poll(): Retrieves and removes the head (first element)

      * of this list.

      */

     System.out.println("Element removed: "+list.poll());
```

```
        // Displaying list elements after poll() operation
        System.out.println("LinkedList after: "+list);
    }
}
```

LinkedList.pollFirst()

public E pollFirst(): Retrieves and removes the first element of this list, or returns null if this list is empty.

```
import java.util.LinkedList;
class LinkedListPollFirstDemo{

    public static void main(String[] args) {

        // Create a LinkedList of Strings
        LinkedList<String> list = new LinkedList<String>();

        // Add few Elements
        list.add("Element1");
        list.add("Element2");
        list.add("Element3");
        list.add("Element4");

        // Display LinkList elements
        System.out.println("LinkedList before: "+list);

        /* pollFirst(): Retrieves and removes the first element
          * of this list, or returns null if this list is empty.
          */

        System.out.println("Element removed: "+list.pollFirst());


        // Display list after calling pollFirst() method

        System.out.println("LinkedList after: "+list);

    }

}

import java.util.LinkedList;

class LinkedListPollLastDemo{


    public static void main(String[] args) {


        // Create a LinkedList of Strings

        LinkedList<String> list = new LinkedList<String>();


        // Add few Elements

        list.add("Element1");

        list.add("Element2");

        list.add("Element3");

        list.add("Element4");


        // Display LinkList elements

        System.out.println("LinkedList before: "+list);


        /* pollFirst(): Retrieves and removes the first element
```

```
         * of this list, or returns null if this list is empty.
         */
        System.out.println("Element removed: "+list.pollLast());

        // Display after calling pollLast() method
        System.out.println("LinkedList after: "+list);
    }
}
```

Java – LinkedList peek(), peekFirst() and peekLast() methods

```
import java.util.LinkedList;
class LinkedListPeekDemo{

    public static void main(String[] args) {

        // Create a LinkedList of Strings
        LinkedList<String> list = new LinkedList<String>();

        // Add few Elements
        list.add("Element1");
        list.add("Element2");
        list.add("Element3");
        list.add("Element4");

        // Display LinkList elements
        System.out.println("LinkedList before: "+list);
```

```
            //peek()

            System.out.println(list.peek());


            //peekFirst()

            System.out.println(list.peekFirst());


            //peekLast()

            System.out.println(list.peekLast());


            // Should be same as peek methods does not remove

            System.out.println("LinkedList after: "+list);

    }

}
```

Java – Convert a LinkedList to ArrayList

```
import java.util.ArrayList;

import java.util.LinkedList;

import java.util.List;


public class ConvertExample {

  public static void main(String[] args) {

        LinkedList<String> linkedlist = new LinkedList<String>();

        linkedlist.add("Harry");

        linkedlist.add("Jack");
        linkedlist.add("Tim");
        linkedlist.add("Rick");
        linkedlist.add("Rock");

        List<String> list = new ArrayList<String>(linkedlist);

        for (String str : list){
            System.out.println(str);
        }
  }
}
```

How to convert LinkedList to array using toArray() in Java

```
import java.util.LinkedList;

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

        //Creating and populating LinkedList
        LinkedList<String> linkedlist = new LinkedList<String>();
        linkedlist.add("Harry");
        linkedlist.add("Maddy");
        linkedlist.add("Chetan");
        linkedlist.add("Chauhan");
        linkedlist.add("Singh");
        //Converting LinkedList to Array

        String[] array = linkedlist.toArray(new String[linkedlist.size()]);


        //Displaying Array content

        System.out.println("Array Elements:");

        for (int i = 0; i < array.length; i++)

        {

              System.out.println(array[i]);

        }

  }

}
```

Vector in Java

Vector implements List Interface. Like ArrayList it also maintains insertion order but it is rarely used in non-thread environment as it is synchronized and due to which it gives poor performance in searching, adding, delete and update of its elements.

Three ways to create vector class object:

Method 1:

```
Vector vec = new Vector();
```

It creates an empty Vector with the default initial capacity of 10. It means the Vector will be re-sized when the 11th elements needs to be inserted into the Vector. Note: By default vector doubles its size. i.e. In this case the Vector size would remain 10 till 10 insertions and once we try to insert the 11th element It would become 20 (double of default capacity 10).

Method 2:

Syntax: Vector object= new Vector(int initialCapacity)

```
Vector vec = new Vector(3);
```

It will create a Vector of initial capacity of 3.

Method 3:

Syntax:

Vector object= new vector(int initialcapacity, capacityIncrement)

Example:

Vector vec= new Vector(4, 6)

Here we have provided two arguments. The initial capacity is 4 and capacityIncrement is 6. It means upon insertion of 5th element the size would be 10 (4+6) and on 11th insertion it would be 16(10+6).

Complete Example of Vector in Java:

```
import java.util.*;


public class VectorExample {


    public static void main(String args[]) {

        /* Vector of initial capacity(size) of 2 */

        Vector<String> vec = new Vector<String>(2);


        /* Adding elements to a vector*/

        vec.addElement("Apple");

        vec.addElement("Orange");

        vec.addElement("Mango");

        vec.addElement("Fig");


        /* check size and capacityIncrement*/

        System.out.println("Size is: "+vec.size());

        System.out.println("Default capacity increment is: "+vec.capacity());


        vec.addElement("fruit1");
```

```
            vec.addElement("fruit2");

            vec.addElement("fruit3");



            /*size and capacityIncrement after two insertions*/

            System.out.println("Size after addition: "+vec.size());

            System.out.println("Capacity after increment is: "+vec.capacity());


            /*Display Vector elements*/

            Enumeration en = vec.elements();

            System.out.println("\nElements are:");

            while(en.hasMoreElements())

                  System.out.print(en.nextElement() + " ");

      }

}
```

How to get sub list of Vector example in java

```
import java.util.Vector;

import java.util.List;


public class SublistExample {


    public static void main(String[] args) {


        // Step 1: Create a Vector

        Vector<String> vector = new Vector<String>();
        // Step 2: Add elements
        vector.add("Item1");
        vector.add("Item2");
        vector.add("Item3");
        vector.add("Item4");
        vector.add("Item5");
        vector.add("Item6");
        /* The method subList(int fromIndex, int toIndex)
          * returns a List having elements of Vector
          * starting from index fromIndex
          * to (toIndex - 1).
          */
        List subList = vector.subList(2,5);

        System.out.println("Sub list elements :");
        for(int i=0; i < subList.size() ; i++){
             System.out.println(subList.get(i));
        }
    }
}
```

How to Sort Vector using Collections.sort in java – Example

Vector maintains the insertion order which means it displays the elements in the same order, in which they got added to the Vector. In this example, we will see how to sort Vector elements in ascending order by using Collections.sort(). The Steps are as follows:

1) Create a Vector object

2) Add elements to the Vector using add(Element e) method

3) Sort it using Collections.sort(Vector object)

4) Display the sorted elements list.

```
import java.util.Collections;

import java.util.Vector;

public class SortingVectorExample {

  public static void main(String[] args) {


        // Create a Vector

        Vector<String> vector = new Vector<String>();


        //Add elements to Vector

        vector.add("Walter");

        vector.add("Anna");

        vector.add("Hank");

        vector.add("Flynn");

        vector.add("Tom");


        // By Default Vector maintains the insertion order

        System.out.println("Vector elements before sorting: ");

        for(int i=0; i < vector.size(); i++){

              //get(i) method fetches the element from index i

              System.out.println(vector.get(i));

        }
        // Collection.sort() sorts the collection in ascending order

        Collections.sort(vector);


        //Display Vector elements after sorting using Collection.sort

        System.out.println("Vector elements after sorting: :");

        for(int i=0; i < vector.size(); i++){

              System.out.println(vector.get(i));

        }

  }

}
```

Search elements in Vector using index – Java example

1) public int indexOf(Object o): It returns the index of first occurrence of Object o in Vector.

2) public int indexOf(Object o, int startIndex): It returns the index of the first occurrence of the Object o in this vector, searching forwards from startIndex (inclusive).

3) public int lastIndexOf(Object o): It returns the index of last occurrence of Object o in Vector.

4) public int lastIndexOf(Object o, int startIndex): It returns the index of the last occurrence of the specified element in this vector, searching backwards from startIndex(inclusive).

```
import java.util.Vector;

public class SearchVector {

    public static void main(String[] args) {

          // Create a Vector object

          Vector<String> vector = new Vector<String>();


          //Add elements to Vector

          vector.add("Kate");

          vector.add("Patt");
```

```
        vector.add("Kluge");

        vector.add("Karon");

        vector.add("Patt");

        vector.add("Monica");

        vector.add("Patt");


        //This would return the index of first occurrence

        int first_index = vector.indexOf("Patt");

        System.out.println("First Occurrence of Patt at index: "+first_index);


        //This would return the index of last occurrence

        int last_index = vector.lastIndexOf("Patt");

        System.out.println("Last Occurrence of Patt at index: "+last_index);


        //This would start search from index 2(inclusive)

        int after_index = vector.indexOf("Patt", 2);

        System.out.println("Occurrence after index 2: "+after_index);


        //This would search the element backward starting from index 6(inclusive)

        int before_index = vector.lastIndexOf("Patt", 6);

        System.out.println("Occurrence before index 6: "+before_index);

    }

}
```

Copy all the elements of one Vector to another Vector example

```
import java.util.Collections;

import java.util.Vector;

public class VectorCopyExample {

    public static void main(String args[])

    {

             //First Vector of String type

             Vector<String> va = new Vector<String>();

             //Adding elements to the first Vector

             va.add("AB");

             va.add("BC");

             va.add("CD");

             va.add("DE");


             //Second Vector

             Vector<String> vb = new Vector<String>();

             //Adding elements to the second Vector

             vb.add("1st");

             vb.add("2nd");

             vb.add("3rd");

             vb.add("4th");

             vb.add("5th");

             vb.add("6th");


             /*Displaying the elements of second vector before

                 performing the copy operation*/
```

```
            System.out.println("Vector vb before copy: "+vb);


            //Copying all the elements of Vector va to Vector vb

            Collections.copy(vb, va);


            //Displaying elements after copy

            System.out.println("Vector vb after copy: "+vb);

      }

}
```

Remove Vector element – Java example

In this example we will see how to remove elements from Vector. We will be using remove(Object o) method of Vector API in order to remove specified elements.

public boolean remove(Object o): Removes the first occurrence of the specified element from Vector If the Vector does not contain the element, it is unchanged.

Example

In this example we are removing two String values from Vector of Strings. The steps are as follows:

1) Create a Vector

2) Add elements to the Vector using add(Element e) method of Vector class.

3) Remove elements using remove(Object o) method of Vector.

```
import java.util.Vector;

public class RemoveFromVector {

    public static void main(String[] args) {

          // Creating a Vector of String Elements
          }

          Vector<String> vector = new Vector<String>();

          //Adding elements to the Vector
          vector.add("Harry");
          vector.add("Steve");
          vector.add("Vince");
          vector.add("David");
          vector.add("Matt");

          System.out.println("Vector elements before remove(): ");
          for(int i=0; i < vector.size(); i++)
          {
              System.out.println(vector.get(i));
          }

          // Removing Harry
          vector.remove("Harry");
          // Removing Matt
          vector.remove("Matt");

          System.out.println("\nVector elements after remove(): ");
          for(int i=0; i < vector.size(); i++)
          {
              System.out.println(vector.get(i));
          }
```

```
    }

}
```

How to remove Vector elements using index in java example

how to remove elements from Vector using index. We will be using remove(int index) method of Vector class.

public E remove(int index): Removes the element at the specified position in this Vector. Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the Vector.

```
import java.util.Vector;

public class RemoveExample

{

  public static void main(String[] args) {


        // Creating a Vector of Strings

        Vector<String> vector = new Vector<String>();


        //Adding elements to the Vector

        vector.add("C++");

        vector.add("Java");

        vector.add("Cobol");

        vector.add("C");

        vector.add("Oracle");


        System.out.println("Vector elements before remove(): ");

        for(int i=0; i < vector.size(); i++)

        {
            System.out.println(vector.get(i));
        }

        // Removing 3rd element from Vector
        Object obj = vector.remove(2);

        System.out.println("\nElement removed from Vector is:");
        System.out.println(obj);

        System.out.println("\nVector elements after remove():");
        for(int i=0; i < vector.size(); i++)
        {
            System.out.println(vector.get(i));
        }
    }
}
```

Remove all elements from Vector in Java – Example

In this example, we will see how to remove all the elements from a Vector. We will be using clear() method of Vector class to do this.

public void clear(): Removes all of the elements from this Vector. The Vector will be empty after this method call.

Example

Here we are displaying the size of the Vector before and after calling clear() method. The steps are as follows:

1) Create a Vector.

2) Add elements to it.

3) Call clear() method to remove all the elements.

```
import java.util.Vector;

public class RemoveAll {

  public static void main(String[] args) {


        // Creating a Vector of Strings

        Vector<String> vector = new Vector<String>();


        //Adding elements to the Vector

        vector.add("C++");

        vector.add("Java");

        vector.add("Cobol");

        vector.add("C");

        vector.add("Oracle");


        System.out.println("Current size of Vector: "+vector.size());


        // Calling clear() method of Vector API

        vector.clear();


        System.out.println("Size of Vector after clear(): "+vector.size());

  }

}
```

Replace Vector elements using index – Java example

In this tutorial, we will see how to replace Vector elements. We will be using set() method of Vector class to do that.

public E set(int index, E element): Replaces the element at the specified position in this Vector with the specified element.

Example

In this example, we are replacing 2nd and 3rd elements of Vector with the new values.

```
import java.util.Vector;

public class ReplaceElements {

  public static void main(String[] args) {

      Vector<String> vector = new Vector<String>();

      vector.add("Harry");

      vector.add("Steve");

      vector.add("Vince");

      vector.add("David");

      vector.add("Matt");


      System.out.println("Vector elements before replacement: ");

      for(int i=0; i < vector.size(); i++)

      {

          System.out.println(vector.get(i));

      }


      //Replacing index 1 element

      vector.set(1,"Mark");
```

```
        //Replacing index 2 element

        vector.set(2,"Jack");


        System.out.println("Vector elements after replacement: ");

        for(int i=0; i < vector.size(); i++)

        {

            System.out.println(vector.get(i));

        }

    }

}
```

How to Set Vector size example

We can set the size of a Vector using setSize() method of Vector class. If new size is greater than the current size then all the elements after current size index have null values. If new size is less than current size then the elements after current size index have been deleted from the Vector.

Example

Here Vector was having 5 elements initially. We set the size to 10 so 5 null elements got inserted at the end of the Vector.

In the second part of the program, we set the size to 4 (less than the current size 10) so last six elements got deleted (including null elements) from Vector.

```
import java.util.Vector;

public class SetSizeExample {

    public static void main(String[] args) {

        // Create a Vector

        Vector<String> vector = new Vector<String>();


        //Add elements to Vector
        vector.add("Walter");
        vector.add("Anna");
        vector.add("Hank");
        vector.add("Flynn");
        vector.add("Tom");

        //Setting up the size greater than current size
        vector.setSize(10);
        System.out.println("Vector size: "+vector.size());
        System.out.println("Vector elements: ");
        for(int i=0; i < vector.size(); i++){
            //get(i) method fetches the element from index i
            System.out.println(vector.get(i));
        }

        //Setting up the size less than current size
        vector.setSize(4);
        System.out.println("\nVector size: "+vector.size());
        System.out.println("Vector elements: ");
        for(int i=0; i < vector.size(); i++){
            System.out.println(vector.get(i));
        }
    }
}
```

Vector Enumeration example in Java

In this example, we are iterating a Vector using Enumeration. The steps are as follows:

1) Create a Vector object

2) Add elements to vector using add() method of Vector class.

3) Call elements() method to get the Enumeration of specified Vector

4) Use hashMoreElements() and nextElement() Methods of Enumeration to iterate through the Vector.

```
import java.util.Vector;

import java.util.Enumeration;


public class VectorEnumerationExample {

  public static void main(String[] args) {

        // Create a Vector

        Vector<String> vector = new Vector<String>();


        // Add elements into Vector

        vector.add("Chaitanya");

        vector.add("Shubham");

        vector.add("Apoorv");

        vector.add("Jin");

        vector.add("Jacob");


        // Get Enumeration of Vector elements

        Enumeration en = vector.elements();


        /* Display Vector elements using hashMoreElements()

          * and nextElement() methods.
```

```
         */
        System.out.println("Vector elements are: ");
        while(en.hasMoreElements())
             System.out.println(en.nextElement());
    }
}
```

Vector Iterator example in Java

In the last tutorial we learnt how to traverse a Vector in both the directions(forward & backward) using ListIterator. In this example, we are gonna see how to traverse a Vector using Iterator. The steps are as follows:

1) Create a Vector

2) Add elements to it using add(Element E) method of Vector class

3) Obtain an iterator by invoking iterator() method of Vector.

4) Traverse the Vector using hasNext() and next() method of Iterator.

```
import java.util.Vector;
import java.util.ListIterator;
import java.util.Iterator;
public class VectorIteratorExample {
    public static void main(String[] args) {

          // Creating a Vector of Strings
          Vector<String> vector = new Vector<String>();

          //Adding elements to the Vector
          vector.add("Mango");
```

```
        vector.add("Orange");

        vector.add("Apple");

        vector.add("Grapes");

        vector.add("Kiwi");


        //Obtaining an iterator

        Iterator it = vector.iterator();


        System.out.println("Vector elements are:");

        while(it.hasNext()){

            System.out.println(it.next());

        }

    }

}
```

Vector ListIterator example in Java

We can traverse a Vector in forward and Backward direction using ListIterator. Along with this we can perform several other operation using methods of ListIterator API like displaying the indexes of next and previous elements, replacing the element value, remove elements during iteration etc.

```
import java.util.Vector;

import java.util.ListIterator;

public class VectorListIteratorDemo {

    public static void main(String[] args) {

        // Create a Vector

        Vector<String> vector = new Vector<String>();
```

```
//Adding elements to the Vector

vector.add("Item1");

vector.add("Item2");

vector.add("Item3");

vector.add("Item4");

vector.add("Item5");


ListIterator litr = vector.listIterator();

System.out.println("Traversing in Forward Direction:");

while(litr.hasNext())

{

    System.out.println(litr.next());
```

```
}

System.out.println("\nTraversing in Backward Direction:");
while(litr.hasPrevious())
{
    System.out.println(litr.previous());
```

```
        }
    }
}
```

Java – Convert Vector to List example

Earlier we shared Vector to ArrayList and Vector to Array conversion. In this tutorial we are gonna see how to convert a Vector to List. The steps are as follows:

1) Create a Vector and populate it

2) Convert it to a List by calling Collections.list(vector.elements()) which returns a List object.

Example

To explain the logic we are assuming that Vector is having elements of String type(). However if you want to have a different type then just change the generics in the below code.

```
import java.util.Vector;

import java.util.List;

import java.util.Collections;

public class VectorToList {


    public static void main(String[] args) {


        // Step1: Creating a Vector of String elements

        Vector<String> vector = new Vector<String>();


        // Step2: Populating Vector

        vector.add("Tim");
```

```
          vector.add("Rock");

          vector.add("Hulk");

          vector.add("Rick");

          vector.add("James");


        // Step3: Displaying Vector elements

        System.out.println("Vector Elements :");

        for (String str : vector){

              System.out.println(str);

        }


        // Step4: Converting Vector to List

        List<String> list = Collections.list(vector.elements());


        // Step 5: Displaying List Elements

        System.out.println("\nList Elements :");

        for (String str2 : list){

              System.out.println(str2);

        }

  }

}
```

Java – Convert Vector to ArrayList example

Example

In the below snippet we have a Vector of Strings and we are converting it to an ArrayList of Strings. The steps we followed in the below example are:

1) Created a Vector and populated it (We are assuming that Vector would be having String elements).

2) Converted the Vector to ArrayList by Declaring ArrayList object using Vector object.

```
ArrayList list = new ArrayList(vector);


import java.util.Vector;

import java.util.ArrayList;

public class VectorToArrayList {


    public static void main(String[] args) {


          // Creating a Vector of String elements

          Vector<String> vector = new Vector<String>();


          // Populate Vector

          vector.add("Rahul");

          vector.add("Steve");

          vector.add("Jude");

          vector.add("Locke");

          vector.add("Mike");

          vector.add("Robert");


          //Displaying Vector elements

          for (String str : vector){

                System.out.println(str);

          }
```

```
        //Converting Vector to ArrayList

        ArrayList<String> arraylist = new ArrayList<String>(vector);


        //Displaying ArrayList Elements

        System.out.println("\nArrayList Elements :");

        for (String s : arraylist){

            System.out.println(s);

        }

    }

}
```

How to convert Vector to String array in java

Couple of weeks back we shared a tutorial on ArrayList to String Array conversion. In this tutorial, we are gonna see how to convert a Vector to String Array in Java.

Example

Lets have a look at the below example where we are converting a Vector of Strings to an array. We are using toString() method of Vector class to do this.

public String toString(): It returns a string representation of this Vector, containing the String representation of each element.

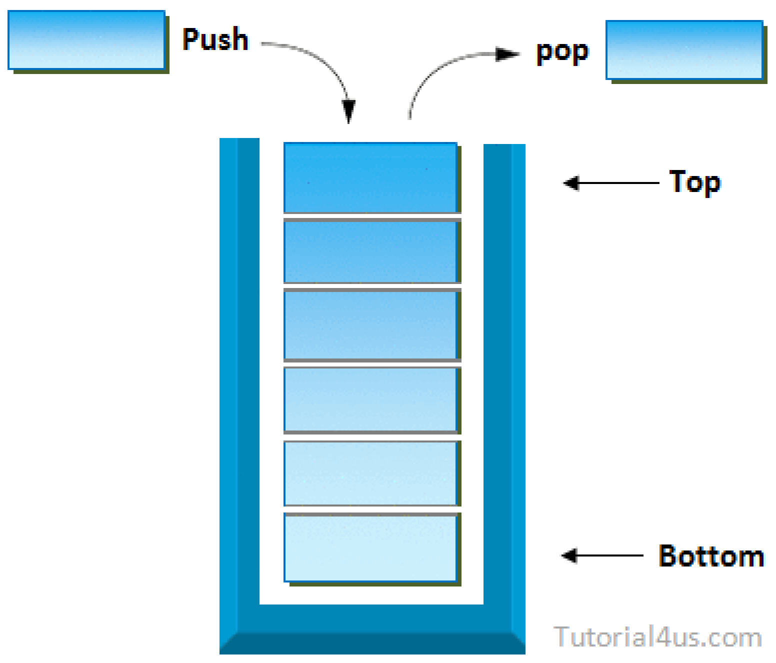
```
import java.util.Vector;

public class VectorToArray {


    public static void main(String[] args) {


        // Creating a Vector of String elements

        Vector<String> vector = new Vector<String>();
```

```
         // Add elements to Vector
         vector.add("Item1");
         vector.add("Item2");
         vector.add("Item3");
         vector.add("Item4");
         vector.add("Item5");
         vector.add("Item6");

        //Converting Vector to String Array
        String[] array = vector.toArray(new String[vector.size()]);

        //Displaying Array Elements
        System.out.println("String Array Elements :");
        for(int i=0; i < array.length ; i++){
           System.out.println(array[i]);
        }
    }
}
```

Stack in Collection Framework:-

Stack is one of the sub-class of Vector class so that all the methods of Vector are inherited into Stack.

The concept of Stack of Data Structure is implemented in java and develop a pre-defined class called Stack.



Stack Important Points

Stack class allow to store Heterogeneous elements.

Stack work on Last in First out (LIFO) manner.

Stack allow to store duplicate values.

Stack class is Synchronized.

Initial 10 memory location is create whenever object of stack is created and it is re-sizable.

Stack also organizes the data in the form of cells like Vector.

Stack is one of the sub-class of Vector.

Creating a Stack is nothing but creating an object of Stack Class.

Syntax

Stack s=new Stack();

Constructors of Stack

Stack(): is used for creating an object of Stack.

Syntax

Stack s=new Stack();

```
}
```

Methods of Stack

public boolean empty(): is used for returns true provided Stack is empty.It returns false in case of Stack is non-empty.

public void push (Object): is used for inserting the elements into the Stack.

public Object pop(): is used for removing Top Most elements from the Stack.

public Object peek(): is used for retrieving Top Most element from the Stack.

public int search(Object): is used for searching an element in the Stack.If the element is found then it returns Stack relative position of that element otherwise it returns -1, -1 indicates search is unsuccessful and element is not found.

Example of Stack add Elements to Stack

```
import java.util.*;

class StackDemo

{

public static void main(String args[])

{

Stack s=new Stack();


//add the data to s

s.push(10);

s.push(20);

s.push(30);

s.push(40);

System.out.println("Stack data: "+s);//    [10,20,30,40]

}

}
```

Example of Stack to add and remove Elements from Stack

```
import java.util.*;

class StackDemo

{

public static void main(String args[])

{

Stack s=new Stack();


// Add the data to s

s.push(10);

s.push(20);

s.push(30);

s.push(40);

System.out.println("Stack elements: "+s);    //[10,20,30,40]


// Remove the top most element

System.out.println("Delete element: "+s.pop());    //40

System.out.println("Stack elements after pop: "+s);//    [10,20,30]

}

}
```

Complete Example of Stack

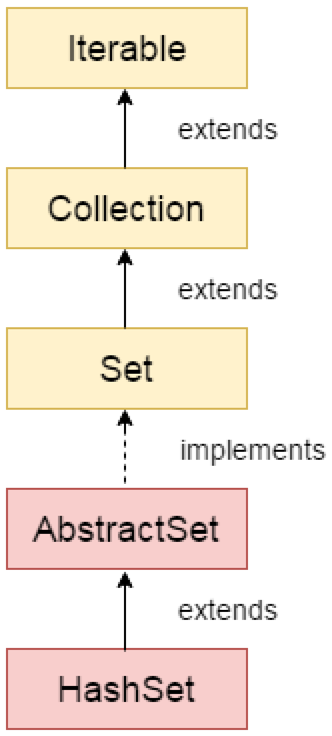
```
import java.util.*;

class StackDemo

{

public static void main(String args[])

{
```

```
Stack s=new Stack();

System.out.println("content of s="+s);        //[]

System.out.println("size of s="+s.size());      //10

System.out.println("Is empty?="s.empty());      //true


//add the data to s

s.push(10);

s.push(20);

s.push(30);

s.push(40);

System.out.println("content of s="+s);    //[10,20,30,40]

System.out.println("size of s="+s.size());    //4

System.out.println("Is s empty ?=s.empty()");    //false

//remove the top most element

System.out.println("delete element="+s.pop());    //40

System.out.println("content of s after pop="+s);//    [10,20,30]

//extract the top most element

System.out.println("top most element="+s.peek());      //30

System.out.println("content of s after peek="+s);              //[10 20 30]

//Search the element 10 and 100

int srp=s.search(10);

  System.out.println("stack relative pos.of 10 is="+srp);          //3

int srp1=s.search(100);

System.out.println("stack relative pos.of 100 is="+srp1);        //-1

}
```

}

Java HashSet:-

Java HashSet class hierarchy



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.

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.

Difference between List and Set

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

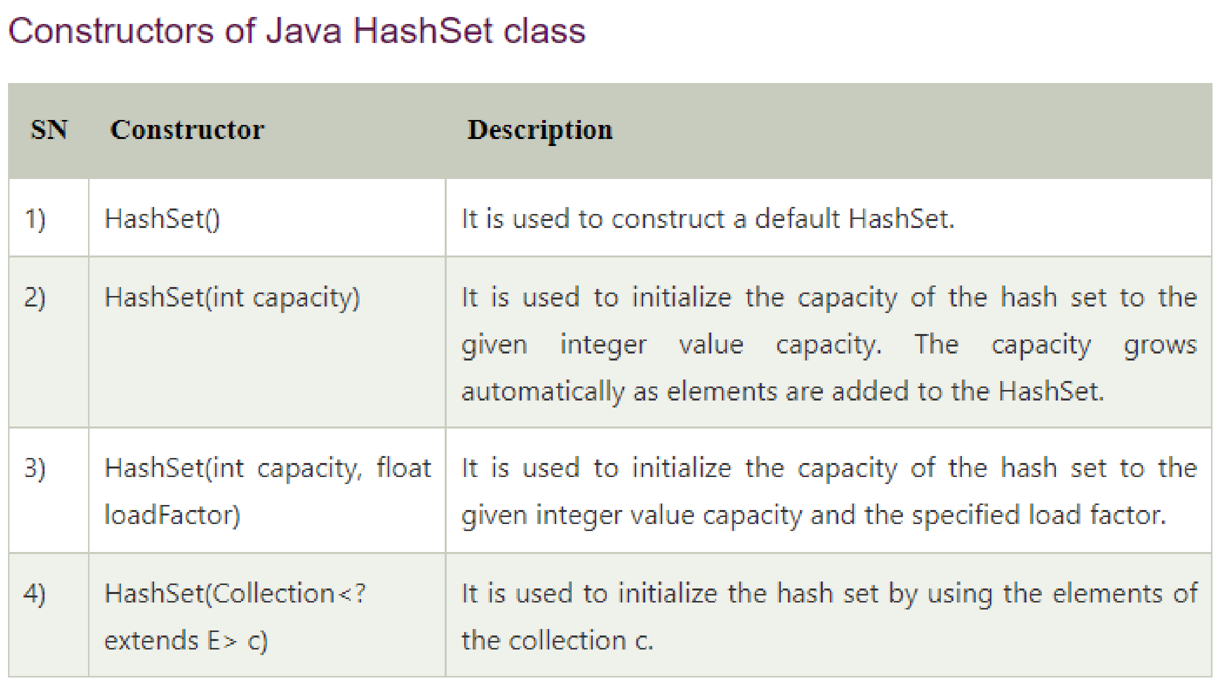
Hierarchy of HashSet class

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.

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

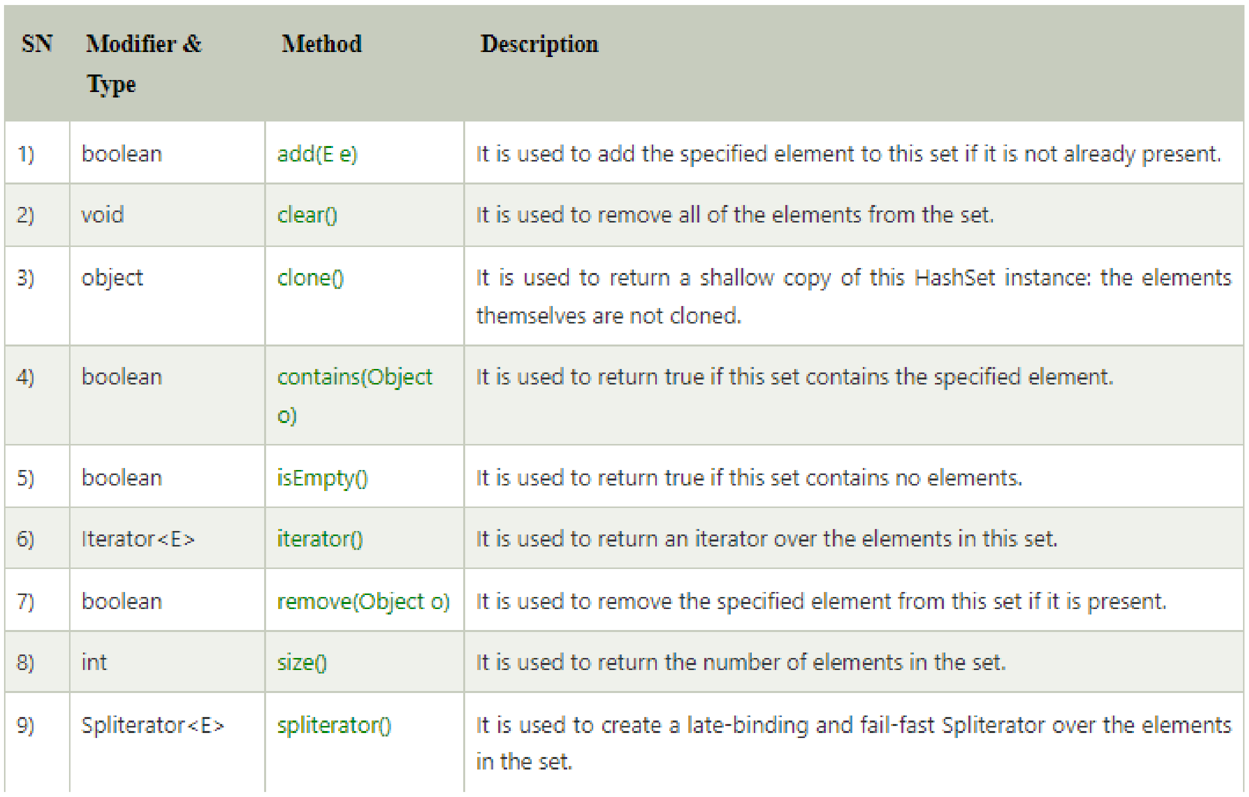


Constructors of Java HashSet class

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, float loadFactor)	It is used to initialize the capacity of the hash set to the given 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.

Methods of Java HashSet class

Various methods of Java HashSet class are as follows:



SN	Modifier & Type	Method	Description
1)	boolean	add(E e)	It is used to add the specified element to this set if it is not already present.
2)	void	clear()	It is used to remove all of the elements from the set.
3)	object	clone()	It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.
4)	boolean	contains(Object o)	It is used to return true if this set contains the specified element.
5)	boolean	isEmpty()	It is used to return true if this set contains no elements.
6)	Iterator<E>	iterator()	It is used to return an iterator over the elements in this set.
7)	boolean	remove(Object o)	It is used to remove the specified element from this set if it is present.
8)	int	size()	It is used to return the number of elements in the set.
9)	Spliterator<E>	spliterator()	It is used to create a late-binding and fail-fast Spliterator over the elements in the set.

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());
                    }
  }
}
```

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());
    }
  }
}
```

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);
  }
}
```

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());
                        }
  }
}
```

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);

        }

}

}
```

Java LinkedHashSet Class

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

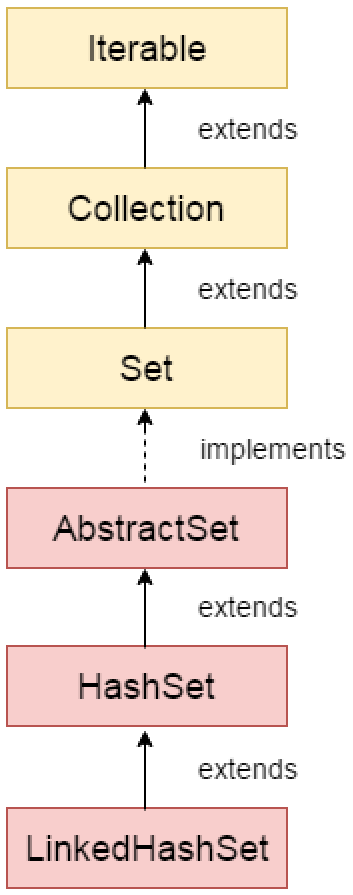
The important points about the Java LinkedHashSet class are:

Java LinkedHashSet class contains unique elements only like HashSet.

Java LinkedHashSet class provides all optional set operations and permits null elements.

Java LinkedHashSet class is non-synchronized.

Java LinkedHashSet class maintains insertion order.



Note: Keeping the insertion order in the LinkedHashset has some additional costs, both in terms of extra memory and extra CPU cycles. Therefore, if it is not required to maintain the insertion order, go for the lighter-weight HashMap or the HashSet instead.

Hierarchy of LinkedHashSet class

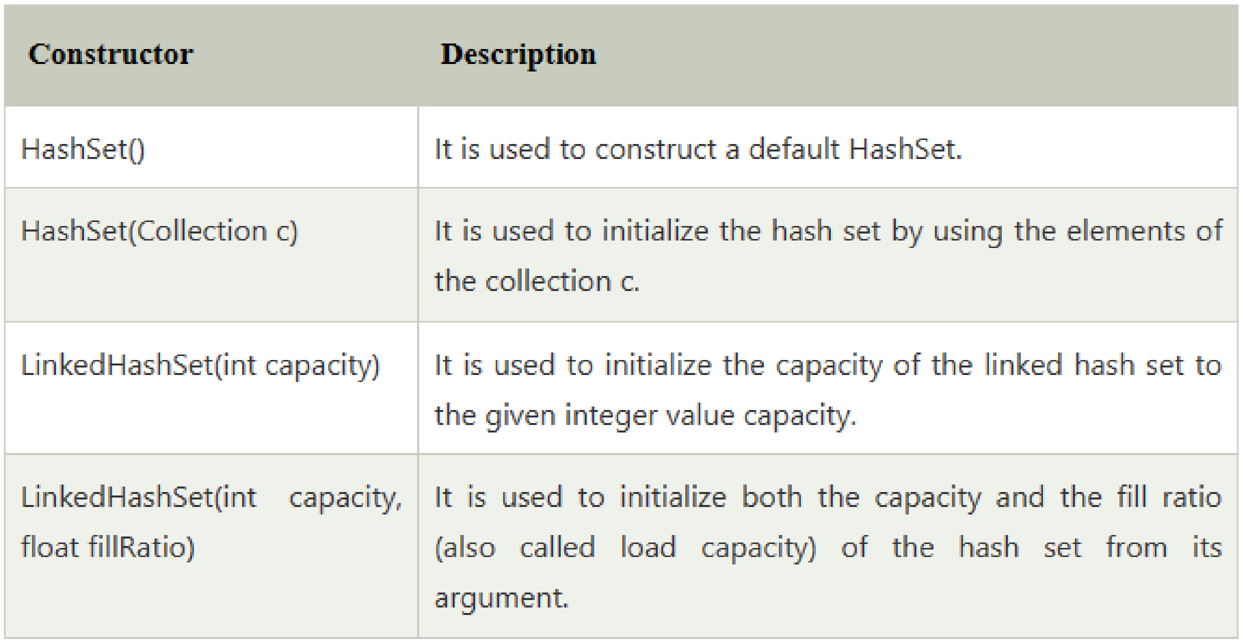
The LinkedHashSet class extends the HashSet class, which implements the Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

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 to 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.

Java LinkedHashSet Example

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

FileName: LinkedHashSet1.java

```
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());
                              }
  }
}
```

Note: We can also use the enhanced for loop for displaying the elements.

Java LinkedHashSet example ignoring duplicate Elements

FileName: LinkedHashSet2.java

```
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());
    }
  }
}
```

Remove Elements Using LinkeHashSet Class

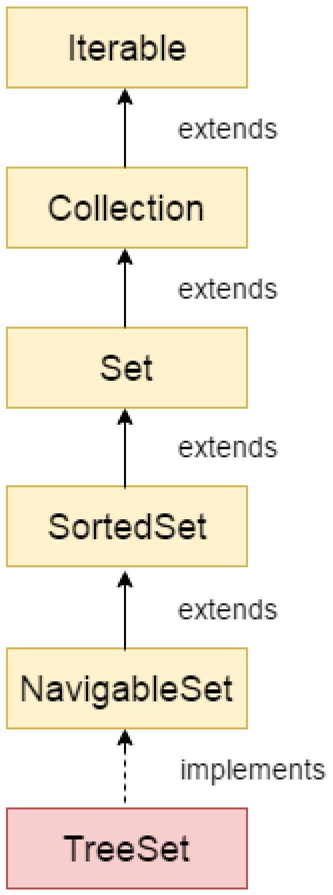
FileName: LinkedHashSet3.java

```
import java.util.*;


public class LinkedHashSet3

{


// main method

public static void main(String argvs[])

{


// Creating an empty LinekdhashSet of string type

LinkedHashSet<String> lhs = new LinkedHashSet<String>();


// Adding elements to the above Set

// by invoking the add() method

lhs.add("Ducat");

lhs.add("India");

lhs.add("Pitampura");

lhs.add("New");

lhs.add("Delhi");


// displaying all the elements on the console

System.out.println("The hash set is: " + lhs);


// Removing an element from the above linked Set
// since the element "Good" is present, therefore, the method remove()

// returns true

System.out.println(lhs.remove("Good"));


// After removing the element

System.out.println("After removing the element, the hash set is: " + lhs);


// since the element "For" is not present, therefore, the method remove()

// returns false

System.out.println(lhs.remove("For"));


}

}
```

Java TreeSet class

TreeSet class hierarchy

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 the Java TreeSet class are:

Java TreeSet class contains unique elements only like HashSet.

Java TreeSet class access and retrieval times are quiet fast.

Java TreeSet class doesn't allow null element.

Java TreeSet class is non synchronized.

Java TreeSet class maintains ascending order.

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 elements.

Java TreeSet class is non-synchronized.

Java TreeSet class maintains ascending order.

The TreeSet can only allow those generic types that are comparable. For example The Comparable interface is being implemented by the StringBuffer class.

Internal Working of The TreeSet Class

TreeSet is being implemented using a binary search tree, which is self-balancing just like a Red-Black Tree. Therefore, operations such as a search, remove, and add consume O(log(N)) time. The reason behind this is there in the self-balancing tree. It is there to ensure that the tree height never exceeds O(log(N)) for all of the mentioned operations. Therefore, it is one of the efficient data structures in order to keep the large data that is sorted and also to do operations on it.

Synchronization of The TreeSet Class

As already mentioned above, the TreeSet class is not synchronized. It means if more than one thread concurrently accesses a tree set, and one of the accessing threads modify it, then the synchronization must be done manually. It is usually done by doing some object synchronization that encapsulates the set. However, in the case where no such object is found, then the set must be wrapped with the help of the Collections.synchronizedSet() method. It is advised to use the method during creation time in order to avoid the unsynchronized access of the set. The following code snippet shows the same.

```
TreeSet treeSet = new TreeSet();

Set syncrSet = Collections.synchronziedSet(treeSet);
```

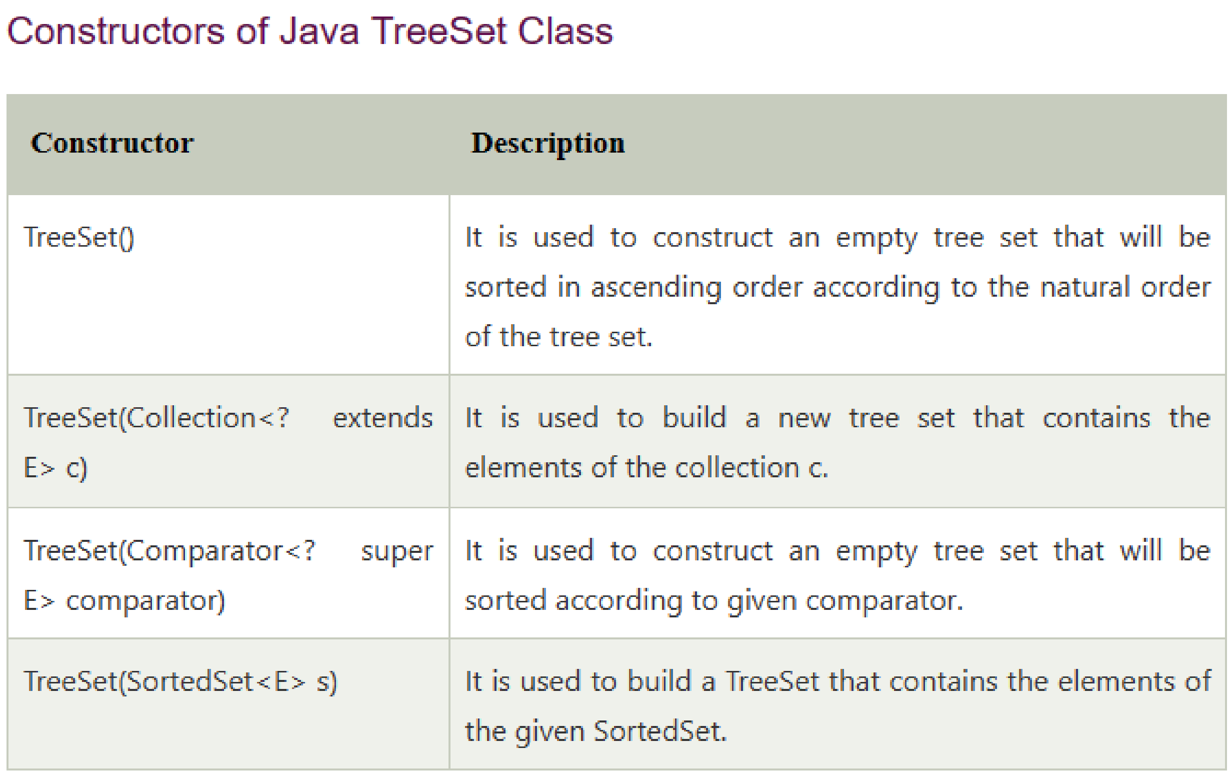
Hierarchy of TreeSet class

As shown in the above diagram, the 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.

Methods of Java TreeSet Class

Java TreeSet Examples

Java TreeSet Example 1:

Let's see a simple example of Java TreeSet.

FileName: TreeSet1.java

```
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());
    }
  }
}
```

Java TreeSet Example 2:

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

FileName: TreeSet2.java

```
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());
                  }


  }

}
```

Java TreeSet Example 3:

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

FileName: TreeSet3.java

```
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("Lowest Value: "+set.pollFirst());

                  System.out.println("Highest Value: "+set.pollLast());

  }

}
```

Java TreeSet Example 4:

In this example, we perform various NavigableSet operations.

FileName: TreeSet4.java

```
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));
  }
}
```

Java TreeSet Example 5:

In this example, we perform various SortedSetSet operations.

FileName: TreeSet5.java

```
import java.util.*;
```

```
class TreeSet5{
  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"));
  }
}
```

Java TreeSet Example: Book

Let's see a TreeSet example where we are adding books to the set and printing all the books. The elements in TreeSet must be of a Comparable type. String and Wrapper classes are Comparable by default. To add user-defined objects in TreeSet, you need to implement the Comparable interface.

FileName: TreeSetExample.java

```
import java.util.*;
class Book implements Comparable<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;

}

// implementing the abstract method

public int compareTo(Book b) {

        if(id>b.id){

                return 1;

        }else if(id<b.id){

                return -1;

        }else{

        return 0;

        }

}

}

public class TreeSetExample {

public static void main(String[] args) {

        Set<Book> set=new TreeSet<Book>();

        //Creating Books

        Book b1=new Book(121,"Let us C","Yashwant Kanetkar","BPB",8);
        Book b2=new Book(233,"Operating System","Galvin","Wiley",6);

        Book b3=new Book(101,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);

        //Adding Books to TreeSet

        set.add(b1);

        set.add(b2);

        set.add(b3);

        //Traversing TreeSet

        for(Book b:set){

        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);

        }

}

}
```

ClassCast Exception in TreeSet

If we add an object of the class that is not implementing the Comparable interface, the ClassCast Exception is raised. Observe the following program.

FileName: ClassCastExceptionTreeSet.java

```
// important import statement

import java.util.*;


class Employee

{


int empId;

String name;
```

```
// getting the name of the employee

String getName()

{

        return this.name;

}


// setting the name of the employee

void setName(String name)

{

this.name = name;

}


// setting the employee id

// of the employee

void setId(int a)

{

this.empId = a;

}


// retrieving the employee id of

// the employee

int getId()

{

return this.empId;
}


}


public class ClassCastExceptionTreeSet

{


// main method

public static void main(String[] argvs)

{

// creating objects of the class Employee

Employee obj1 = new Employee();


Employee obj2 = new Employee();


TreeSet<Employee> ts =    new TreeSet<Employee>();


// adding the employee objects to

// the TreeSet class

ts.add(obj1);

ts.add(obj2);


System.out.println("The program has been executed successfully.");


}
```

```
}
```

HashMap in Java with Example

HashMap is a Map based collection class that is used for storing Key & value pairs, it is denoted as HashMap<Key, Value> or HashMap<K, V>. This class makes no guarantees as to the order of the map. It is similar to the Hashtable class except that it is unsynchronized and permits nulls(null values and null key).

It is not an ordered collection which means it does not return the keys and values in the same order in which they have been inserted into the HashMap. It does not sort the stored keys and Values. You must need to import java.util.HashMap or its super class in order to use the HashMap class and methods.

HashMap Example in Java:

In this example we have demonstrated almost all the important methods of HashMap class.

```
import java.util.HashMap;

import java.util.Map;

import java.util.Iterator;

import java.util.Set;

public class Details {


    public static void main(String args[]) {


        /* This is how to declare HashMap */

        HashMap<Integer, String> hmap = new HashMap<Integer, String>();
```

```
/*Adding elements to HashMap*/

hmap.put(12, "Chaitanya");

hmap.put(2, "Rahul");

hmap.put(7, "Singh");

hmap.put(49, "Ajeet");

hmap.put(3, "Anuj");


/* Display content using Iterator*/

Set set = hmap.entrySet();

Iterator iterator = set.iterator();

while(iterator.hasNext()) {

    Map.Entry mentry = (Map.Entry)iterator.next();

    System.out.print("key is: "+ mentry.getKey() + " & Value is: ");

    System.out.println(mentry.getValue());

}


/* Get values based on key*/

String var= hmap.get(2);

System.out.println("Value at index 2 is: "+var);


/* Remove values based on key*/

hmap.remove(3);

System.out.println("Map key and values after removal:");

Set set2 = hmap.entrySet();

Iterator iterator2 = set2.iterator();
```

```
            while(iterator2.hasNext()) {

                Map.Entry mentry2 = (Map.Entry)iterator2.next();

                System.out.print("Key is: "+mentry2.getKey() + " & Value is: ");

                System.out.println(mentry2.getValue());

             }


        }

}
```

How to sort HashMap in Java by Keys and Values

As we know that HashMap doesn’t preserve any order by default. If there is a need we need to sort it explicitly based on the requirement. In this tutorial we will learn how to sort HashMap by keys using TreeMap and by values using Comparator.

HashMap Sorting by Keys

In this example we are sorting the HashMap based on the keys using the TreeMap collection class.

```
import java.util.HashMap;

import java.util.Map;

import java.util.TreeMap;

import java.util.Set;

import java.util.Iterator;


public class Details {


    public static void main(String[] args) {


         HashMap<Integer, String> hmap = new HashMap<Integer, String>();
}

hmap.put(5, "A");

hmap.put(11, "C");

hmap.put(4, "Z");

hmap.put(77, "Y");

hmap.put(9, "P");

hmap.put(66, "Q");

hmap.put(0, "R");


System.out.println("Before Sorting:");

Set set = hmap.entrySet();

Iterator iterator = set.iterator();

while(iterator.hasNext()) {

            Map.Entry me = (Map.Entry)iterator.next();

            System.out.print(me.getKey() + ": ");

            System.out.println(me.getValue());

}

Map<Integer, String> map = new TreeMap<Integer, String>(hmap);

System.out.println("After Sorting:");

Set set2 = map.entrySet();

Iterator iterator2 = set2.iterator();

while(iterator2.hasNext()) {

          Map.Entry me2 = (Map.Entry)iterator2.next();

          System.out.print(me2.getKey() + ": ");

          System.out.println(me2.getValue());

}
```

```
        }
}
```

Java – Get size of HashMap example

In this example we are gonna see how to get the size of HashMap using size() method of HashMap class. Method definition and description are as follows:

public int size(): Returns the number of key-value mappings in this map.

```
import java.util.HashMap;

public class SizeExample {

  public static void main(String[] args) {

        // Creating a HashMap of int keys and String values
        HashMap<Integer, String> hashmap = new HashMap<Integer, String>();

        // Adding Key and Value pairs to HashMap
        hashmap.put(11,"Value1");
        hashmap.put(22,"Value2");
        hashmap.put(33,"Value3");
        hashmap.put(44,"Value4");
        hashmap.put(55,"Value5");

        // int size() method returns the number of key value pairs
        System.out.println("Size of HashMap : " + hashmap.size());
  }
}
```

Java – Remove mapping from HashMap example

Example

In this example we are gonna see how to remove a specific mapping from HashMap using the key value of Key-value pair. We will be using the following method of HashMap class to perform this operation:

public Value remove(Object key): Removes the mapping for the specified key from this map if present and returns the Element value for that particular Key. More about remove method from Javadoc.

```
import java.util.HashMap;

public class RemoveMappingExample {

  public static void main(String[] args) {

        // Creating a HashMap of int keys and String values
        HashMap<Integer, String> hashmap = new HashMap<Integer, String>();

        // Adding Key and Value pairs to HashMap
        hashmap.put(11,"Value1");
        hashmap.put(22,"Value2");
        hashmap.put(33,"Value3");
        hashmap.put(44,"Value4");
        hashmap.put(55,"Value5");
        hashmap.put(66,"Value6");
        // Displaying HashMap Elements

        System.out.println("HashMap Elements: " + hashmap);


        // Removing Key-Value pairs for key 33

        Object removedElement1 = hashmap.remove(33);

        System.out.println("Element removed is: " +removedElement1);


        // Removing Key-Value pairs for key 55

        Object removedElement2 = hashmap.remove(55);

        System.out.println("Element removed is: " +removedElement2);


        // Displaying HashMap Elements after remove

        System.out.println("After Remove:");

        System.out.println("--------------");

        System.out.println("HashMap Elements: " + hashmap);

  }

}
```

Java – Remove all mappings from HashMap example

we shared how to remove a specific mapping from HashMap based on key. In this example we are going to see how to remove all the mappings from HashMap. We will be using clear() method of HashMap class to do this:

public void clear(): Removes all of the mappings from this map. The map will be empty after this call returns.

```
import java.util.HashMap;
```

```
}

public class RemoveAllExample {

  public static void main(String[] args) {

        // Creating a HashMap of int keys and String values
        HashMap<Integer, String> hashmap = new HashMap<Integer, String>();

        // Adding Key and Value pairs to HashMap
        hashmap.put(11,"Value1");
        hashmap.put(22,"Value2");
        hashmap.put(33,"Value3");
        hashmap.put(44,"Value4");
        hashmap.put(55,"Value5");

        // Displaying HashMap Elements
        System.out.println("HashMap Elements: " + hashmap);

        // Removing all Mapping
        hashmap.clear();

        // Displaying HashMap Elements after remove
        System.out.println("After calling clear():");
        System.out.println("---------------------");
        System.out.println("HashMap Elements: " + hashmap);
  }
```

```
}
```

How to check if a HashMap is empty or not?

Description

Program to check if a HashMap is empty or not. We are using isEmpty() method of HashMap class to perform this check.

```
import java.util.HashMap;

class HashMapIsEmptyExample{


  public static void main(String args[]) {


    // Create a HashMap

    HashMap<Integer, String> hmap = new HashMap<Integer, String>();




    // Checking whether HashMap is empty or not

    /* isEmpty() method signature and description -

     * public boolean isEmpty(): Returns true if this map

     * contains no key-value mappings.

     */

    System.out.println("Is HashMap Empty? "+hmap.isEmpty());


    // Adding few elements

    hmap.put(11, "Jack");

    hmap.put(22, "Rock");

    hmap.put(33, "Rick");
```

```
        hmap.put(44, "Smith");

        hmap.put(55, "Will");


        // Checking again

        System.out.println("Is HashMap Empty? "+hmap.isEmpty());

    }

}
```

Java – Check if a particular key exists in HashMap example

we learnt how to check whether a particular value exists in HashMap. In this example we are gonna see how to check if a particular key is present in HashMap. We will be using containsKey() method of HashMap class to perform this check. The method definition and description are as follows:

public boolean containsKey(Object key): Returns true if this map contains a mapping for the specified key.

Example

The steps we followed in the below example are:

1) Create a HashMap and populate it with key-value pairs.

2) Check any key existence by calling containsKey() method. This method returns a boolean value.

```
import java.util.HashMap;


public class CheckKeyExample {


  public static void main(String[] args) {
        // Creating a HashMap of int keys and String values
        HashMap<Integer, String> hashmap = new HashMap<Integer, String>();

        // Adding Key and Value pairs to HashMap
        hashmap.put(11,"Chaitanya");
        hashmap.put(22,"Pratap");
        hashmap.put(33,"Singh");
        hashmap.put(44,"Rajesh");
        hashmap.put(55,"Kate");

        // Checking Key Existence
        boolean flag = hashmap.containsKey(22);
        System.out.println("Key 22 exists in HashMap? : " + flag);

        boolean flag2 = hashmap.containsKey(55);
        System.out.println("Key 55 exists in HashMap? : " + flag2);

        boolean flag3 = hashmap.containsKey(99);
        System.out.println("Key 99 exists in HashMap? : " + flag3);
  }
}
```

Java – Check if a particular value exists in HashMap example

we are checking whether a particular value exists in HashMap or not. We will be using containsValue() method of HashMap class to perform this check:

public boolean containsValue(Object value): Returns true if this map maps one or more keys to the specified value.

```
import java.util.HashMap;


public class CheckValueExample {


  public static void main(String[] args) {


        // Creating a HashMap of int keys and String values

        HashMap<Integer, String> hashmap = new HashMap<Integer, String>();


        // Adding Key and Value pairs to HashMap

        hashmap.put(11,"Chaitanya");

        hashmap.put(22,"Pratap");

        hashmap.put(33,"Singh");

        hashmap.put(44,"Rajesh");

        hashmap.put(55,"Kate");


        // Checking Value Existence

        boolean flag = hashmap.containsValue("Singh");

        System.out.println("String Singh exists in HashMap? : " + flag);

  }

}
```

How to serialize HashMap in java

HashMap class is serialized by default which means we need not to implement Serializable interface in order to make it eligible for Serialization. In this tutorial we will learn How to write HashMap object and it’s content into a file and How to read the HashMap object from the file. Before I share the complete code for this let me give a brief info about Serialization and De-serialization.

Serialization: It is a process of writing an Object into file along with its attributes and content. It internally converts the object in stream of bytes.

De-Serialization: It is a process of reading the Object and it's properties from a file along with the Object's content.

Example:

Serialization of HashMap: In the below class we are storing the HashMap content in a hashmap.ser serialized file. Once you run the below code it would produce a hashmap.ser file. This file would be used in the next class for de-serialization.

```
import java.io.*;

import java.util.HashMap;

public class Details

{

        public static void main(String [] args)

        {

                HashMap<Integer, String> hmap = new HashMap<Integer, String>();

                //Adding elements to HashMap

                hmap.put(11, "AB");

                hmap.put(2, "CD");

                hmap.put(33, "EF");

                hmap.put(9, "GH");

                hmap.put(3, "IJ");

                try

                {
                    FileOutputStream fos =
                          new FileOutputStream("hashmap.ser");
                    ObjectOutputStream oos = new ObjectOutputStream(fos);
                    oos.writeObject(hmap);
                    oos.close();
                    fos.close();
                    System.out.printf("Serialized HashMap data is saved in hashmap.ser");
             }catch(IOException ioe)
              {
                    ioe.printStackTrace();
              }
        }
}
```

How to synchronize HashMap in Java with example

HashMap is a non-synchronized collection class. If we need to perform thread-safe operations on it then we must need to synchronize it explicitly. In this tutorial we will see how to synchronize HashMap.

Example:

In this example we have a HashMap<Integer, String> it is having integer keys and String type values. In order to synchronize it we are using Collections.synchronizedMap(hashmap)   it returns a thread-safe map backed up by the specified HashMap.

Important point to note in the below example:

Iterator should be used in a synchronized block even if we have synchronized the HashMap explicitly (As we did in the below code).

Syntax:

```
Map map = Collections.synchronizedMap(new HashMap());

...

//This doesn't need to be in synchronized block

Set set = map.keySet();

// Synchronizing on map, not on set

synchronized (map) {

            // Iterator must be in synchronized block

            Iterator iterator = set.iterator();

            while (iterator.hasNext()){

                    ...

            }

}import java.util.Collections;

import java.util.HashMap;

import java.util.Map;

import java.util.Set;

import java.util.Iterator;

public class HashMapSyncExample {

        public static void main(String args[]) {

                HashMap<Integer, String> hmap= new HashMap<Integer, String>();

                hmap.put(2, "Anil");

                hmap.put(44, "Ajit");

                hmap.put(1, "Brad");

                hmap.put(4, "Sachin");

                hmap.put(88, "XYZ");
            Map map= Collections.synchronizedMap(hmap);

            Set set = map.entrySet();

            synchronized(map)

         {

                Iterator i = set.iterator();

                // Display elements

                while(i.hasNext()) {

                     Map.Entry me = (Map.Entry)i.next();

                     System.out.print(me.getKey() + ": ");

                     System.out.println(me.getValue());

                }

            }

      }

}
```

TreeSet Class in Java with example

TreeSet is similar to HashSet except that it sorts the elements in the ascending order while HashSet doesn’t maintain any order. TreeSet allows null element but like HashSet it doesn’t allow. Like most of the other collection classes this class is also not synchronized, however it can be synchronized explicitly like this: SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));

TreeSet Example:

In this example we have two TreeSet (TreeSet<String> & TreeSet<Integer>). We have added the values to both of them randomly however the result we got is sorted in ascending order.

```
import java.util.TreeSet;

public class TreeSetExample {

     public static void main(String args[]) {

            // TreeSet of String Type
```

```
        TreeSet<String> tset = new TreeSet<String>();


        // Adding elements to TreeSet<String>

        tset.add("ABC");

        tset.add("String");

        tset.add("Test");

        tset.add("Pen");

        tset.add("Ink");

        tset.add("Jack");


        //Displaying TreeSet

        System.out.println(tset);


        // TreeSet of Integer Type

        TreeSet<Integer> tset2 = new TreeSet<Integer>();


        // Adding elements to TreeSet<Integer>

        tset2.add(88);

        tset2.add(7);

        tset2.add(101);

        tset2.add(0);

        tset2.add(3);

        tset2.add(222);

        System.out.println(tset2);

    }
```

```
  }
```

How to convert a HashSet to a TreeSet

```
import java.util.HashSet;

import java.util.TreeSet;

import java.util.Set;

class ConvertHashSettoTreeSet{

   public static void main(String[] args) {

        // Create a HashSet

        HashSet<String> hset = new HashSet<String>();


        //add elements to HashSet

        hset.add("Element1");

        hset.add("Element2");

        hset.add("Element3");

        hset.add("Element4");


        // Displaying HashSet elements

        System.out.println("HashSet contains: "+ hset);


        // Creating a TreeSet of HashSet elements

        Set<String> tset = new TreeSet<String>(hset);


        // Displaying TreeSet elements

        System.out.println("TreeSet contains: ");
        for(String temp : tset){
              System.out.println(temp);
        }
   }
}
```

LinkedHashSet Class in Java with Example

Earlier we have shared tutorials on HashSet and TreeSet. LinkedHashSet is also an implementation of Set interface, it is similar to the HashSet and TreeSet except the below mentioned differences:

HashSet doesn’t maintain any kind of order of its elements.

TreeSet sorts the elements in ascending order.

LinkedHashSet maintains the insertion order. Elements gets sorted in the same sequence in which they have been added to the Set.

```
import java.util.LinkedHashSet;
public class LinkedHashSetExample {
        public static void main(String args[]) {
                // LinkedHashSet of String Type
                LinkedHashSet<String> lhset = new LinkedHashSet<String>();

                // Adding elements to the LinkedHashSet
                lhset.add("Z");
                lhset.add("PQ");
                lhset.add("N");
                lhset.add("O");
                lhset.add("KK");
                lhset.add("FGH");
                System.out.println(lhset);


                // LinkedHashSet of Integer Type

                LinkedHashSet<Integer> lhset2 = new LinkedHashSet<Integer>();


                // Adding elements

                lhset2.add(99);

                lhset2.add(7);

                lhset2.add(0);

                lhset2.add(67);

                lhset2.add(89);

                lhset2.add(66);

                System.out.println(lhset2);

        }

}
```

Queue Interface in Java Collections

A Queue is designed in such a way so that the elements added to it are placed at the end of Queue and removed from the beginning of Queue. The concept here is similar to the queue we see in our daily life, for example, when a new iPhone launches we stand in a queue outside the apple store, whoever is added to the queue has to stand at the end of it and persons are served on the basis of FIFO (First In First Out), The one who gets the iPhone is removed from the beginning of the queue.

Queue Interface Java Hierarchy

Queue interface in Java collections has two implementation: LinkedList and PriorityQueue, these two classes implements Queue interface.

Queue is an interface so we cannot instantiate it, rather we create instance of LinkedList or PriorityQueue and assign it to the Queue like this:

```
Queue q1 = new LinkedList();
Queue q2 = new PriorityQueue();

import java.util.*;

public class QueueExample1 {


    public static void main(String[] args) {


        /*

          * We cannot create instance of a Queue as it is an

          * interface, we can create instance of LinkedList or

          * PriorityQueue and assign it to Queue

          */

        Queue<String> q = new LinkedList<String>();


        //Adding elements to the Queue

        q.add("Rick");

        q.add("Maggie");

        q.add("Glenn");

        q.add("Negan");

        q.add("Daryl");


        System.out.println("Elements in Queue:"+q);


        /*

          * We can remove element from Queue using remove() method,

          * this would remove the first element from the Queue
          */

        System.out.println("Removed element: "+q.remove());


        /*

          * element() method - this returns the head of the

          * Queue. Head is the first element of Queue

          */

        System.out.println("Head: "+q.element());


        /*

          * poll() method - this removes and returns the

          * head of the Queue. Returns null if the Queue is empty

          */

        System.out.println("poll(): "+q.poll());


        /*

          * peek() method - it works same as element() method,

          * however it returns null if the Queue is empty

          */

        System.out.println("peek(): "+q.peek());


        //Again displaying the elements of Queue

        System.out.println("Elements in Queue:"+q);

    }

}
```

PriorityQueue Interface in Java Collections

we have seen how a Queue serves the requests based on FIFO(First in First out). Now the question is: What if we want to serve the request based on the priority rather than FIFO? In a practical scenario this type of solution would be preferred as it is more dynamic and efficient in nature. This can be done with the help of PriorityQueue, which serves the request based on the priority that we set using Comparator.

Java PriorityQueue Example

In this example, I am adding few Strings to the PriorityQueue, while creating PriorityQueue I have passed the Comparator(named as MyComparator) to the PriorityQueue constructor.

In the MyComparator java class, I have sorted the Strings based on their length, which means the priority that I have set in PriorityQueue is String length. That way I ensured that the smallest string would be served first rather than the string that I have added first.

```
import java.util.PriorityQueue;

public class PriorityQueueExample

{

    public static void main(String[] args)

    {


        PriorityQueue<String> queue =

            new PriorityQueue<String>(15, new MyComparator());

        queue.add("Tyrion Lannister");

        queue.add("Daenerys Targaryen");

        queue.add("Arya Stark");

        queue.add("Petyr 'Littlefinger' Baelish");


        /*

         * What I am doing here is removing the highest

         * priority element from Queue and displaying it.
```

```
          * The priority I have set is based on the string
          * length. The logic for it is written in Comparator
          */
         while (queue.size() != 0)
         {
             System.out.println(queue.poll());

         }
     }
}
```

MyComparator.java

```
import java.util.Comparator;

public class MyComparator implements Comparator<String>
{
   @Override
   public int compare(String x, String y)
   {
      return x.length() - y.length();
   }
}
```