Collections Framework

Table of Contents

1. Collection Overview

Collections in java is a framework that provides an architecture to store and manipulate the group of objects.

All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

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

What is Collection in java ?

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

What is framework in java ?

- provides readymade architecture.
- represents set of classes and interface.
- is optional.

What is Collection framework ?

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

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

Advantages of Collection Framework:

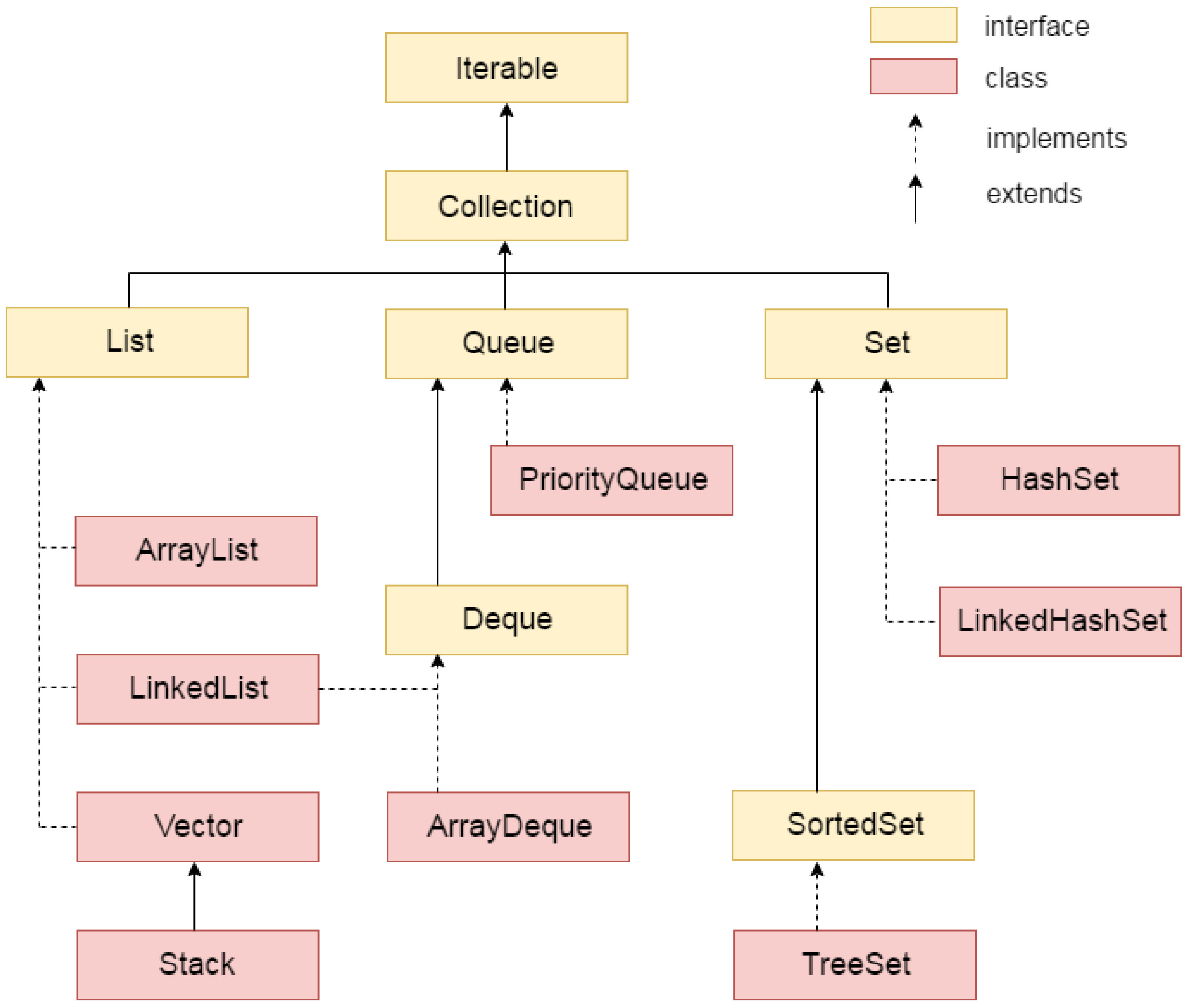
1.Consistent API : The API has basic set of interfaces like Collection, Set, List, or Map. All those classes (such as ArrayList, LinkedList, Vector etc) which implements, these interfaces have some common set of methods.

2.Reduces programming effort: The programmer need not to worry about design of Collection rather than he can focus on its best use in his program.

3.Increases program speed and quality: Increases performance by providing high-performance implementations of useful data structures and algorithms.

Hierarchy of Collection Framework

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



2. Iterators

Iterators are used in Collection framework in Java to retrieve elements one by one. There are three iterators.

Enumeration

- It is a interface used to get elements of legacy collections(Vector, Hashtable).
- Enumeration is the first iterator present from JDK 1.0, rests are included in JDK 1.2 with more functionality.
- We can create Enumeration object by calling ***elements()*** method of vector class on any vector object.

```
// Here "v" is an Vector class object. e is of
// type Enumeration interface and refers to "v"
Enumeration e = v.elements();
```

This iterator is based on data from Enumeration and has methods:

- **booleanhasMoreElements()**: This method tests if the Enumeration has any more elements or not .
- **element nextElement()**: This returns the next element that is available in elements that is available in Enumeration

```
import java.util.Vector;
import java.util.Enumeration;
public class Test
{
   public static void main(String args[])
   {
      Vector dayNames = new Vector();
      dayNames.add("Sunday");
      dayNames.add("Monday");
      dayNames.add("Tuesday");
      dayNames.add("Wednesday");
      dayNames.add("Thursday");
      dayNames.add("Friday");
      dayNames.add("Saturday");

      // Creating enumeration
      Enumeration days = dayNames.elements();

      // Retrieving elements of enumeration
      while (days.hasMoreElements())
         System.out.println(days.nextElement());
   }
}
```

Output:

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

Limitations of Enumeration :

- Enumeration is for **legacy** classes(Vector, Hashtable) only. Hence it is not a universal iterator.
- Remove operations can't be performed using Enumeration.
- Only forward direction iterating is possible.

Iterator

- It is a **universal** iterator as we can apply it to any Collection object. By using Iterator, we can perform both read and remove operations.
- It is improved version of Enumeration with additional functionality of remove-ability of a element.
- Iterator must be used whenever we want to enumerate elements in all Collection framework implemented interfaces like Set, List, Queue, Deque and also in all implemented classes of Map interface.
- Iterator is the **only** cursor available for entire collection framework.
- Iterator object can be created by calling ***iterator()*** method present in Collection interface.

```
// Here "c" is any Collection object.

// itr is of  type Iterator interface and refers to "c"

Iterator itr = c.iterator();
```

It has 3 methods:

- **boolean hasNext():** This method returns true if the iterator has more elements.
- **elements next()**: This method returns the next elements in the iterator. It throws NoSuchElementException if no more element present.
- **void remove()**: This method removes from the collection the last elements returned by the iterator.

```
// Java program to demonstrate working of iterators
import java.util.*;
public class IteratorDemo
{
   public static void main(String args[])
   {
      //create a Hashset to store strings
      HashSet<String> hs = new HashSet<String>() ;

      // store some string elements
      hs.add("India");
      hs.add ("America");
      hs.add("Japan");

      // Add an Iterator to hs.
      Iterator it = hs.iterator();

      // Display element by element using Iterator
```

```
        while (it.hasNext())
            System.out.println(it.next());
    }
}
```

Output:

```
America
Japan
India
```

Limitations of Iterator :

- Only forward direction iterating is possible.
- Replacement and addition of new element is not supported by Iterator.

ListIterator

- It is only applicable for List collection implemented classes like arraylist, linkedlist etc.
- It is an interface that contains methods to retrieve the elements from a collection object, both in **forward and reverse directions**.
- ListIterator must be used when we want to enumerate elements of List. This cursor has more functionality(methods) than iterator.

ListIterator object can be created by calling ***listIterator()*** method present in List interface.

```
// Here "l" is any List object, ltr is of type
// ListIterator interface and refers to "l"
ListIterator ltr = l.listIterator();
```

- ListIterator interface extends Iterator interface. So all three methods of Iterator interface are available for ListIterator. In addition there are **some** more methods.

- **booleanhasNext():** This returns true if the ListIterator has more elements when traversing the list in the forward direction.
- **booleanhasPerivous():** This returns true if the ListIterator has more elements when traversing the list in the reverse direction.
- **element next():** This returns the next element in the list.
- **element previous():** This returns the previous element in the list.
- **void remove():** This removes from the list the last elements that was returned by the next() or previous() methods.
- **int nextIndex():** Returns the index of the element that would be returned by a subsequent call to next(). (Returns list size if the list iterator is at the end of the list.)
- **int previousIndex():** Returns the index of the element that would be returned by a subsequent call to previous(). (Returns -1 if the list iterator is at the beginning of the list.)

```
// Java program to demonstrate working of ListIterator
import java. util.* ;

class Test
{
    public static void main(String args[])
    {
        // take a vector to store Integer objects
        Vector<Integer> v = new Vector<Integer>();

        // Adding Elements to Vector
        v.add(10);
        v.add(20);
        v.add(30);

        // Creating a ListIterator
        ListIterator lit = v.listIterator();
        System.out.println("In Forward direction:");

        while (lit.hasNext())
            System.out.print(lit.next()+" ") ;

        System.out.print("\n\nIn backward direction:\n") ;
        while (lit.hasPrevious())
            System.out.print(lit.previous()+" ");
    }
}
```

Output :

```
In Forward direction:
10 20 30


In backward direction:
30 20 10
```

Limitations of ListIterator :

It is the most powerful iterator but it is only applicable for List implemented classes, so it is not a universal iterator.

Q. How is Iterator different from ListIterator?

- Iterator can retrieve the elements only in forward direction. But ListIterator can retrieve the elements in forward and reverse direction also. So ListIterator is preferred to Iterator.

- Using ListIterator, we can get iterator’s current position.
- we can use LisIterator with vector and list (e.g. ArrayList).
- Since ListIterator can access elements in both directions and supports additional operators, ListIterator cannot be applied on Set (e.g., HashSet and TreeSet.)

  Exm:

```
import java.util.*;
public class IteratorDemo
{
    public static void main(String args[])
    {
        //create a Hashset to store strings
        HashSet<String> hs = new HashSet<String>() ;

        // store some string elements
        hs.add("India");
        hs.add ("America");
        hs.add("Japan");

        // Add an Iterator to hs.
        Listlterator it = hs.listlterator();

        // Display element by element using Iterator
        System.out.println( "Elements using Iterator: ");
        while (it.hasNext())
            System.out.println(it.next());
    }
}
```

Output:

```
prog.java:15: error: cannot find symbol
        Listlterator it = hs.listlterator();
        ^
  symbol:   class Listlterator
  location: class IteratorDemo
prog.java:15: error: cannot find symbol
        Listlterator it = hs.listlterator();
                            ^
  symbol:   method listIterator()
  location: variable hs of type HashSet<String>
2 errors
```

Q. What is the difference between Iterator and Enumeration?

- Iterator has an option to remove elements from the collection which is not available in Enumeration.
- Also, Iterator has methods whose names are easy to follow and Enumeration methods are difficult to remember.

3. Iterator vs Foreach

Iterator is an interface provided by collection framework to traverse a collection and for a sequential access of items in the collection.

```
// Iterating over collection 'c' using Iterator
for (Iterator i = c.iterator(); i.hasNext(); )
  System.out.println(i.next());
```

For each loop is meant for traversing items in a collection.

```
// Iterating over collection 'c' using for-each
for (Element e: c)
  System.out.println(e);
```

We read the **':'** used in for-each loop as **"in".** So loop reads as "for each element e in elements", here elements is the collection which stores Element type items.

Note : In Java 8 using lambda expressions we can simply replace for-each loop with

```
elements.forEach (e -> System.out.println(e) );
```

Difference between the two traversals

In for-each loop, we can't modify collection, it will throw a **ConcurrentModificationException** on the other hand with iterator we can modify collection.

- Modifying a collection simply means removing an element or changing content of an item stored in the collection.

- This occurs because for-each loop implicitly creates an iterator but it is not exposed to the user thus we can't modify the items in the collections.

When to use which traversal?

- If we have to modify collection, we must use an Iterator.
- While using nested for loops it is better to use for-each loop, consider the below code for better understanding.

```
// Java program to demonstrate working of nested iterators
// may not work as expected and throw exception.
import java.util.*;

public class Main
{
    public static void main(String args[])
    {
        // Create a link list which stores integer elements
        List<Integer> l = new LinkedList<Integer>();
       // Now add elements to the Link List
        l.add(2);
        l.add(3);
        l.add(4);
       // Make another Link List which stores integer elements
        List<Integer> s=new LinkedList<Integer>();
        s.add(7);
        s.add(8);
        s.add(9);

        // Iterator to iterate over a Link List
        for (Iterator<Integer> itr1=l.iterator(); itr1.hasNext(); )
        {
            for (Iterator<Integer> itr2=s.iterator(); itr2.hasNext(); )
            {
                if (itr1.next() < itr2.next())
                {
                    System.out.println(itr1.next());
                }
            }
        }
    }
}
```

Output:

```
3
Exception in thread "main" java.util.NoSuchElementException
            at java.util.LinkedList$ListItr.next(LinkedList.java:888)
            at Main.main(Main.java:29)
```

The above code throws java.util.NoSuchElementException.

- In the above code we are calling the next() method again and again for itr1 (i.e., for List l).
- Now we are advancing the iterator without even checking if it has any more elements left in the collection(in the inner loop), thus we are advancing the iterator more than the number of elements in the collection which leads to NoSuchElementException.

for-each loops are tailor made for nested loops. Replace the iterator code with the below code.

```
// Java program to demonstrate working of nested for-each
import java.util.*;
public class Main
{
    public static void main(String args[])
    {
        // Create a link list which stores integer elements
        List<Integer> l=new LinkedList<Integer>();
        // Now add elements to the Link List
        l.add(2);
        l.add(3);
        l.add(4);

        // Make another Link List which stores integer elements
        List<Integer> s=new LinkedList<Integer>();
        s.add(2);
        s.add(4);
        s.add(5);
        s.add(6);
        // Iterator to iterate over a Link List
        for (int a:l)
        {
           for (int b:s)
           {
              if (a<b)
                 System.out.print(a + " ");
           }
        }
    }
}
```

Output:

```
2 2 2 3 3 3 4 4
```

Performance Analysis

- Traversing a collection using for-each loops or iterators give the same performance. Here, by performance we mean the time complexity of both these traversals.
- If you iterate using the old styled C for loop then we might increase the time complexity drastically.

```
// Here l is List ,it can be ArrayList /LinkedList and n is size of the List
```

```
for (i=0;i<n;i++)
  System.out.println(l.get(i));
```

- Here if the list l is an ArrayList then we can access it in **O(1)** time since it is allocated contiguous memory blocks (just like an array) i.e random access is possible.
- But if the collection is LinkedList, then random access is not possible since it is not allocated contiguous memory blocks, so in order to access a element we will have to traverse the link list till you get to the required index, thus the time taken in worst case to access an element will be **O(n).**

Iterator and for-each loop are faster than simple for loop for collections with no random access, while in collections which allows random access there is no performance change with for-each loop/for loop/iterator.

4. List Interface

- List is an ordered collection of objects in which duplicate values can be stored.
- Since List preserves the insertion order it allows positional access and insertion of elements.
- List Interface is implemented by ArrayList, LinkedList, Vector and Stack classes.

Creating List Objects:

List is an interface, we can create instance of List in following ways:

```
List a = new ArrayList();

List b = new LinkedList();

List c = new Vector();

List d = new Stack();
```

Generic List Object:

After the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the List. We can declare type safe List in following way:

```
// Obj is type of object to be stored in List.

List<Obj> list = new List<Obj> ();
```

Operations on List:

List Interface extends Collection, hence it supports all the operations of Collection Interface and along with following operations:

1. Positional Access:

List allows add, remove, get and set operations based on numerical positions of elements in List. List provides following methods for these operations:

- **void add(int index,Object O):** This method adds given element at specified index.
- **boolean addAll(int index, Collection c):** This method adds all elements from specified collection to list. First element gets inserted at given index. If there is already an element at that position, that element and other subsequent elements(if any) are shifted to the right by increasing their index.
- **Object remove(int index):** This method removes an element from the specified index. It shifts subsequent elements(if any) to left and decreases their indexes by 1.
- **Object get(int index):** This method returns element at the specified index.
- **Object set(int index, Object new):** This method replaces element at given index with new element. This function returns the element which was just replaced by new element.

```
// Java program to demonstrate positional access

// operations on List interface

import java.util.*;
```

```
 public class ListDemo
{
    public static void main (String[] args)
    {
        // Let us create a list
        List l1 = new ArrayList();
        l1.add(0, 1);  // adds 1 at 0 index
        l1.add(1, 2);  // adds 2 at 1 index
        System.out.println(l1);  // [1, 2]
        // Let us create another list
        List l2 = new ArrayList();
        l2.add(1);
        l2.add(2);
        l2.add(3);
        // will add list l2 from 1 index
        l1.addAll(1, l2);
        System.out.println(l1); //[1,1,2,3,2]

        l1.remove(1);     // remove element from index 1
        System.out.println(l1); // [1, 2, 3, 2]

        // prints element at index 3
        System.out.println(l1.get(3)); // 2

        l1.set(0, 5);   // replace 0th element with 5
        System.out.println(l1);  // [5, 2, 3, 2]
    }
}
```

Output :

```
[1, 2]
[1, 1, 2, 3, 2]
[1, 2, 3, 2]
2
[5, 2, 3, 2]
```

2. **Search:**

List provides methods to search element and returns its numeric position. Following two methods are supported by List for this operation:

- **int indexOf(Object o):** This method returns first occurrence of given element or -1 if element is not present in list.
- **int lastIndexOf(Object o):** This method returns the last occurrence of given element or -1 if element is not present in list.

```
// Java program to demonstrate search
// operations on List interface
import java.util.*;

public class ListDemo
{
   public static void main(String[] args)
   {
      // type safe array list, stores only string
      List<String> l = new ArrayList<String>(5);
      l.add("kaliber");
      l.add("mind");
      l.add("Academy");

      // Using indexOf() and lastIndexOf()
      System.out.println("first index of Kaliber:" +l.indexOf("kaliber"));
      System.out.println("last index of Academy:" + l.lastIndexOf("Academy"));
      System.out.println("Index of element not present : " + l.indexOf("Hello"));
   }
}
```

Output :

```
first index of Kaliber: 0
last index of Academy: 2
Index of element not present : -1
```

3. **Iteration:**

ListIterator(extends Iterator) is used to iterate over List element. List iterator is bidirectional iterator.

4. Range-view:

List Interface provides method to get List view of the portion of given List between two indices. Following is the method supported by List for range view operation.

- **List subList(int fromIndex,int toIndex):** This method returns List view of specified List between fromIndex(inclusive) and toIndex(exclusive).

```
// Java program to demonstrate subList operation
// on List interface.
import java.util.*;
public class ListDemo
{
   public static void main (String[] args)
   {
      // Type safe array list, stores only string
      List<String> l = new ArrayList<String>(5);

      l.add("KaliberMind");
      l.add("Practice");
      l.add("KaliberQuiz");
      l.add("JAVA");
      l.add("Courses");

      List<String> range = new ArrayList<String>();

      // return List between 2nd(including)
      // and 4th element(excluding)
      range = l.subList(2, 4);

      System.out.println(range);  // [KaliberQuiz, JAVA]
   }
}
```

Output :

```
[KaliberQuiz, JAVA]
```

4(a). ArrayList class

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The issue with Array is that they are of fixed length so if it is full you can not resize it and add any more elements to it, likewise if there are number of elements gets removed from it the memory consumption would be the same as it doesn't shrink.

On the other hand ArrayList can dynamically grow and shrink after addition and removal operations.

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- ArrayList can not be used for primitive types, like int, char, etc. We need a wrapper class for such cases.
- In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

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 collection. Now it is type safe so typecasting is not required at run time.

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

1. ArrayList al=**new** ArrayList();//creating old non-generic arraylist

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

1. ArrayList<String> al=**new** ArrayList<String>();//creating new generic arraylist

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

ArrayList Example

```
import java.util.*;

 public class ArrayListExample {
   public static void main(String args[]) {
        /*Declaration of ArrayList */
  ArrayList<String> al = new ArrayList<String>();

  /*Elements  added to the ArrayList*/
  al.add("Apple");
  al.add("Pear");
  al.add("Banana");
  al.add("PineApple");
  al.add("Orange");

  /* Showing ArrayList elements */
  System.out.println("ArrayList contains: "+al);

  /*Add element at the given index*/
  al.add(0, "Blackberry");
  al.add(1, "Kiwi");

  /*Remove elements from array list like this*/
  al.remove("Pear");
  al.remove("Orange");

  System.out.println("Current ArrayList is: "+al);

  /*Remove element from the given index*/
  al.remove(1);

  System.out.println("Updated ArrayList is:"+al);
 }
}
```

Output :

```
ArrayList contains: [Apple, Pear, Banana, PineApple, Orange]
Current ArrayList is: [Blackberry, Kiwi, Apple, Banana, PineApple]
Updated ArrayList is:[Blackberry, Apple, Banana, PineApple]
```

ArrayList Basics

1.Initialize ArrayList

2 Ways On How To Initialize An Arraylist In Java With Example :

Method 1: Using Arrays.asList() method

Syntax :

```
ArrayList<Type> obj = new ArrayList<Type>(
       Arrays.asList(Object o1, Object o2, Object o3, ....so on));
```

Example:

```
import java.util.*;

 public class Initialization1 {
   public static void main(String args[]) {
      ArrayList<String> obj = new ArrayList<String>(
   Arrays.asList("Boston", "Chicago", "Dallas"));
      System.out.println("Elements are:"+obj);
  }
}
```

Output :

```
Elements are:[Boston, Chicago, Dallas]
```

Method 2: Normal Initialization method

Syntax :

```
ArrayList<T> obj = new ArrayList<T>();
   obj.add("Object o1");
   obj.add("Object o2");
   obj.add("Object o3");
                    ...
                    ...
```

Example:

```
import java.util.*;

 public class Initialization2 {
   public static void main(String args[]) {
      ArrayList<String> cars = new ArrayList<String>();
   cars.add("Honda");
   cars.add("Hyundai");
   cars.add("Toyota");
   System.out.println("Cars stored in array list are: "+ cars);
  }
}
```

Output :

```
Cars stored in array list are: [Honda, Hyundai, Toyota]
```

2. Iterate (Loop) ArrayList

There are 5 ways you can iterate through the ArrayList.

1. For Loop
2. Advanced For Loop
3. Iterator
4. While loop
5. Collection's stream() util (Java 8)

Lets write all the ways through which one can iterate or traverse or loop over ArrayList in java.

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


public class ArrayListLoopExample
{
 public static void main (String[] args)
 {
  ArrayList<Integer> al = new ArrayList<Integer>();
  al.add(13);
  al.add(7);
  al.add(36);
  al.add(89);
  al.add(97);


      /*For Loop for traversing ArrayList*/
      System.out.println("Using For Loop");
      for (int counter = 0; counter < al.size(); counter++) {
        System.out.println(al.get(counter));
      }
```

```
    /* Advanced For Loop for Iterating ArrayList*/
    System.out.println("Using Advanced For Loop");
    for (Integer num : al) {
        System.out.println(num);
    }

    /* While Loop for iterating ArrayList*/
    System.out.println("Using While Loop");
    int count = 0;
    while (al.size() > count) {
  System.out.println(al.get(count));
      count++;
    }

    /*Traversing ArrayList using Iterator*/
    System.out.println("Using Iterator");
    Iterator it = al.iterator();
    while (it.hasNext()) {
       System.out.println(it.next());
    }
    /* Iterating ArrayList using Collection stream() util */
    System.out.println("Using Collection stream() util");
    al.forEach((num) -> {
    System.out.println(num);
    });

}
}
```

Output

```
Using For Loop
13
7
36
89
97
Using Advanced For Loop
13
7
36
89
```

```
97
Using While Loop
13
7
36
89
97
Using Iterator
13
7
36
89
97
Using Collection stream() util
13
7
36
89
97
```

3. <u>Find Length of ArrayList</u>

❖ How To Find Length/Size Of An ArrayList In Java With Example ?

- One can use the size() method of java.util.ArrayList to find the length or size of ArrayList in java.
- The size() method returns an integer equal to a number of elements present in the ArrayList.
- ArrayList is dynamic in nature. It is different than the length of the Array which is backing the ArrayList that is called the capacity of ArrayList.
- When you create an object of ArrayList without specifying a capacity, it is created with default capacity which is 10.
- Since ArrayList is dynamic in nature, it automatically grows in size when the number of element in the ArrayList reaches a threshold.
- When ArrayList is first created it is known as empty ArrayList which has size() of zero.If you add elements one by one then size grows one by one.

ArrayList size() example in Java

size() method returns the number of elements present in ArrayList but it is different than capacity which is equal to the length of the underlying array and always greater than the size.

❖ **Java Program to find Number of Elements in ArrayList**

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

/* Write a program to determine the size/length of the ArrayList*/
public class ArrayListSize
{
   public static void main (String[] args)
   {
      ArrayList<Integer> al=new ArrayList<Integer>();
      System.out.println("Size before adding elements: "+al.size());
      al.add(11);
      al.add(3);
      al.add(5);
      al.add(4);
      al.add(9);
      System.out.println("Size after adding elements: "+al.size());
      al.remove(1);
      al.remove(2);
      System.out.println("Size after remove operations: "+al.size());
      System.out.println("Final ArrayList: ");
      for(int num: al){
         System.out.println(num);
      }
   }
}
```

Output :

```
Size before adding elements: 0
Size after adding elements: 5
Size after remove operations: 3
Final ArrayList:
11
5
9
```

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

```
/* Program to find size of ArrayList in Java */
public class ArrayListSize
{
      public static void main (String[] args)
  {
      System.out.println("Java Program to find the length of array list");
      ArrayList<String> listOfCities = new ArrayList<>();
      int size = listOfCities.size();
      System.out.println("size of array list after creating: " + size);
      listOfCities.add("California");
      listOfCities.add("Boston");
      listOfCities.add("New York");
      size = listOfCities.size();
      System.out.println("length of ArrayList after adding elements: " + size);
      listOfCities.clear();
      size = listOfCities.size();
      System.out.println("size of ArrayList after clearing elements: " + size);
  }
}
```

Output :

```
Java Program to find the length of array list
size of array list after creating: 0
length of ArrayList after adding elements: 3
size of ArrayList after clearing elements: 0
```

ArrayList Sorting

1. <u>Sort ArrayList in Ascending Order</u>

How To Sort An ArrayList In Java With Example ?

- **Program for Sorting String ArrayList :**

  Here we are sorting the ArrayList of String type. We will use the Collections.sort(list) method. In this case the output list will be sorted alphabetically.

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

 public static void main(String args[]){
   ArrayList<String> listofcountries = new ArrayList<String>();
   listofcountries.add("USA");
   listofcountries.add("UK");
   listofcountries.add("India");
   listofcountries.add("Canada");
   /*Unsorted List*/
   System.out.println("Before Sorting:");
   for(String counter: listofcountries){
  System.out.println(counter);
 }

   /* Sort statement*/
   Collections.sort(listofcountries);

   /* Sorted List*/
   System.out.println("After Sorting:");
   for(String counter: listofcountries){
  System.out.println(counter);
 }
 }
}
```

Output :

```
Before Sorting:
USA
UK
India
Canada
After Sorting:
Canada
India
UK
USA
```

- **Program for Sorting Integer ArrayList :**

  We will use the same Collections.sort(list) method to sort the Integer ArrayList.

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

 public static void main(String args[]){
    ArrayList<Integer> arraylist = new ArrayList<Integer>();
    arraylist.add(5);
    arraylist.add(7);
    arraylist.add(8);
    arraylist.add(3);
    /* ArrayList before the sorting*/
    System.out.println("Before Sorting:");
    for(int counter: arraylist){
   System.out.println(counter);
  }

    /* Sorting of arraylist using Collections.sort*/
    Collections.sort(arraylist);

    /* ArrayList after sorting*/
    System.out.println("After Sorting:");
    for(int counter: arraylist){
   System.out.println(counter);
  }
 }
}
```

Output:

```
Before Sorting:
5
7
8
3
After Sorting:
3
5
7
8
```

2. Sort ArrayList in Descending Order

To sort the ArrayList in descending order we will use two methods **Collections.reverseOrder()** method and **Collections.sort()** method.

- One liner will be like :
  Collections.sort(arraylist, Collections.reverseOrder());

- Another way of sorting ArrayList in Descending order is to sort the list in ascending order first and then it will be reversed.
  Collections.sort(list);
  Collections.reverse(list);

❖ **Code of Sorting ArrayList in Descending Order**

```
import java.util.*;

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

      ArrayList<String> arraylist = new ArrayList<String>();
   arraylist.add("Apple");
   arraylist.add("Banana");
   arraylist.add("Pear");
   arraylist.add("Mango");

   /*Unsorted List: ArrayList content before sorting*/
   System.out.println("ArrayList Before Sorting:");
   for(String str: arraylist){
  System.out.println(str);
  }

   /* Sorting in decreasing (descending) order*/
   Collections.sort(arraylist, Collections.reverseOrder());

   /* Sorted List in reverse order*/
   System.out.println("ArrayList in descending order:");
   for(String str: arraylist){
  System.out.println(str);
    }
  }
}
```

Output :

```
ArrayList Before Sorting:
Apple
Banana
Pear
Mango
ArrayList in descending order:
```

```
Pear
Mango
Banana
Apple
```

- Above example is to sort String ArrayList. Similarly, we can sort ArrayList of Integers in descending order.

```
import java.util.*;

 public class ArrayListDescendingSort2 {
   public static void main(String args[]) {

      ArrayList<Integer> arraylist = new ArrayList<Integer>();
   arraylist.add(1);
   arraylist.add(13);
   arraylist.add(89);
   arraylist.add(45);

   /*Unsorted List: ArrayList content before sorting*/
   System.out.println("ArrayList Before Sorting:");
   for(int num: arraylist){
  System.out.println(num);
  }

   /* Sorting in decreasing (descending) order*/
   Collections.sort(arraylist, Collections.reverseOrder());

   /* Sorted List in reverse order*/
   System.out.println("ArrayList in descending order:");
   for(int num: arraylist){
  System.out.println(num);
  }
  }
}
```

Output :

```
ArrayList Before Sorting:
1
13
89
45
ArrayList in descending order:
```

```
89
45
13
1
```

3. Sort ArrayList of Objects using Comparable and Comparator

- We have already seen how to sort ArrayList in ascending order. In this contents we will see how to sort an ArrayList of Objects by property using Comparable and Comparator interface.
- Comparable and Comparator interfaces are used if the ArrayList is of custom object type.

Why do we need Comparable and Comparator?

In the below example we have a Student class which has properties like roll no.,Student name and Student age.

```
public class Student  {
   private String studentname;
   private int rollno;
   private int studentage;

   public Student(int rollno, String studentname, int studentage) {
        this.rollno = rollno;
        this.studentname = studentname;
        this.studentage = studentage;
   }

   public String getStudentname() {
        return studentname;
   }
   public void setStudentname(String studentname) {
 this.studentname = studentname;
   }
   public int getRollno() {
 return rollno;
   }
   public void setRollno(int rollno) {
```

```
 this.rollno = rollno;
   }
   public int getStudentage() {
 return studentage;
   }
   public void setStudentage(int studentage) {
  this.studentage = studentage;
   }
}
```

We will add Students to the ArrayList object.

```
import java.util.*;

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

      ArrayList<Student> arraylist = new ArrayList<Student>();
   arraylist.add(new Student(111, "John", 23));
   arraylist.add(new Student(222, "Messi", 29));
   arraylist.add(new Student(333, "Ronaldo", 31));

   Collections.sort(arraylist);

   for(Student str: arraylist){
  System.out.println(str);
   }
  }
}
```

We tried to call the **Collections.sort()** method on List of Objects and got the following error message.
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
Bound mismatch: The generic method sort(List) of type Collections is not applicable for the arguments (ArrayList). The inferred type Student is not a valid substitute for the bounded parameter > at ArrayListSort.main(ArrayListSort.java:15)

Reason : I just called the sort method on an ArrayList of Objects which doesn't work until unless we use interfaces like Comparable and Comparator. Now we will use Comparable and Comparator to get the sorting done in our way.

Sorting of ArrayList(Object) Using Comparable

Suppose we need to sort the ArrayList object based on **student Age** property.

To achieve this we will first implement **Comparable interface** and then override the **compareTo()** method.

```
public class Student implements Comparable {
   private String studentname;
   private int rollno;
   private int studentage;

   public Student(int rollno, String studentname, int studentage) {
      this.rollno = rollno;
      this.studentname = studentname;
      this.studentage = studentage;
   }
   //getter and setter methods same as the above example
   @Override
   public int compareTo(Student comparestu) {
      int compareage=((Student)comparestu).getStudentage();
      /* For Ascending order*/
      return this.studentage-compareage;
     /* For Descending order do like this */
      //return compareage-this.studentage;
   }
@Override
   public String toString() {
      return "[ rollno=" + rollno + ", name=" + studentname + ", age=" + studentage + "]";
   }
}
```

Now we can call Collections.sort() on ArrayList

```
import java.util.*;

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

       ArrayList<Student> arraylist = new ArrayList<Student>();
    arraylist.add(new Student(222, "Messi", 29));
    arraylist.add(new Student(333, "Ronaldo", 31));
    arraylist.add(new Student(111, "john", 23));

    Collections.sort(arraylist);

    for(Student str: arraylist){
   System.out.println(str);
    }
     }
}
```

Output :

```
[ rollno=111, name=John, age=23]
[ rollno=222, name=Messi, age=29]
[ rollno=333, name=Ronaldo, age=31]
```

Q. Why do we need Comparator when we Already have Comparable ?

If you want to have more than one way of sorting your class, you must implement Comparator.

Sorting ArrayList(Object) Multiple Properties Using Comparator

To implement Comparator we need to override compare method for sorting.

```
import java.util.Comparator;
public class Student  {
    private String studentname;
    private int rollno;
```

```
  private int studentage;

  public Student(int rollno, String studentname, int studentage) {
     this.rollno = rollno;
     this.studentname = studentname;
     this.studentage = studentage;
  }
  ...
  //Getter and setter methods same as the above examples
  ...
  /*Comparator for sorting the list by Student Name*/
  public static Comparator<Student> StuNameComparator = new Comparator<Student>() {

public int compare(Student s1, Student s2) {
  String StudentName1 = s1.getStudentname().toUpperCase();
  String StudentName2 = s2.getStudentname().toUpperCase();

  //ascending order
  return StudentName1.compareTo(StudentName2);

  //descending order
  //return StudentName2.compareTo(StudentName1);
  }};

  /*Comparator for sorting the list by roll no*/
  public static Comparator<Student> StuRollno = new Comparator<Student>() {

  public int compare(Student s1, Student s2) {

  int rollno1 = s1.getRollno();
  int rollno2 = s2.getRollno();
```

```
   /*For ascending order*/
   return rollno1-rollno2;

   /*For descending order*/
   //rollno2-rollno1;
  }};
   @Override
   public String toString() {
      return "[ rollno=" + rollno + ", name=" + studentname + ", age=" + studentage + "]";
   }
```

ArrayList Class

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

 public static void main(String args[]){
   ArrayList<Student> arraylist = new ArrayList<Student>();
   arraylist.add(new Student(111, "John", 30));
   arraylist.add(new Student(333, "Ronaldo", 31));
   arraylist.add(new Student(222, "Messi", 29));

   /*Sorting based on Student Name*/
   System.out.println("Student Name Sorting:");
   Collections.sort(arraylist, Student.StuNameComparator);

   for(Student str: arraylist){
  System.out.println(str);
   }
 /* Sorting on Rollno property*/
   System.out.println("RollNum Sorting:");
   Collections.sort(arraylist, Student.StuRollno);
   for(Student str: arraylist){
```

```
  System.out.println(str);
   }
 }
}
```
Output :

```
Student Name Sorting:
[ rollno=111, name=John, age=30]
[ rollno=222, name=Messi, age=29]
[ rollno=333, name=Ronaldo, age=31]
RollNum Sorting:
[ rollno=222, name=Messi, age=29]
[ rollno=111, name=John, age=30]
[ rollno=333, name=Ronaldo, age=31]
```

ArrayList Add/Remove

1. Add element to ArrayList

This is a concept for adding an element to the ArrayList in java. The method signature of add method is

public boolean add(Object element)

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


      //ArrayList<String> Declaration
      ArrayList<String> al= new ArrayList<String>();
      //add method for String ArrayList
      al.add("California");
      al.add("New York");
      al.add("Boston");
      al.add("San jose");
      al.add("San Francisco");
      System.out.println("Elements of ArrayList of String Type: "+al);

      //ArrayList<Integer> Declaration
      ArrayList<Integer> arrlist = new ArrayList<Integer>();
      //add method for Integer ArrayList
```

```
        arrlist.add(12);
        arrlist.add(3);
        arrlist.add(9);
        arrlist.add(96);
        arrlist.add(8);
        System.out.println("Elements of ArrayList of Integer Type: "+arrlist);
    }
}
```

Output :

```
Elements of ArrayList of String Type: [California, New York, Boston, San jose, San Francisco]
Elements of ArrayList of Integer Type: [12, 3, 9, 96, 8]
```

2. Add element at Particular Index of ArrayList

Previously I have shown how to add element at the end of the list in java by using add() method. There is another variant of add method which adds element at the specified index.The syntax for the method is

public void add(int index, Object element)

```
import java.util.*;
public class AddMethodExample {
    public static void main(String args[]) {
        // ArrayList of String type
       ArrayList<String> al = new ArrayList<String>();
       // simple add() methods for adding elements at the end
       al.add("California");
       al.add("Boston");
       al.add("San jose");
       al.add("New York");
       //adding element to the 3rd position
       //3rd position = 2 index as index starts with 0
       al.add(2,"San Francisco");
       System.out.println("Elements after adding string San Francisco:"+ al);
       //adding string to 1st position
       al.add(0, "Texas");

       //Print
       System.out.println("Elements after adding string Texas:"+ al);
    }
}
```

Output :

```
Elements after adding string San Francisco:[California, Boston, San Francisco, San jose, New York]
Elements after adding string Texas:[Texas, California, Boston, San Francisco, San jose, New York]
```

3. Append Collection elements to ArrayList

In this concepts I will share how to use addAll(Collection c) method of java.util.ArrayList class.

The syntax of the addAll method is :

public boolean addAll(Collection c)

```java
import java.util.*;
public class AddAllMethodExample {
   public static void main(String args[]) {
      // ArrayList1 of String type
      ArrayList<String> al = new ArrayList<String>();
      al.add("California");
      al.add("Illinois");
      al.add("New York");
      al.add("Texas");
      System.out.println("ArrayList1 before addAll:"+al);
      //ArrayList2 of String Type
      ArrayList<String> al2 = new ArrayList<String>();
      al2.add("Text1");
      al2.add("Text2");
      al2.add("Text3");
      al2.add("Text4");

      //Adding ArrayList2 into ArrayList1
      al.addAll(al2);
      System.out.println("ArrayList1 after addAll:"+al);
   }
}
```

Output :

```
ArrayList1 before addAll:[California, Illinois, New York, Texas]
ArrayList1 after addAll:[California, Illinois, New York, Texas, Text1, Text2, Text3, Text4]
```

4. Insert all the Collection elements at the Specified Position in ArrayList

In the last tutorial I have shared an example of addAll(Collection c) method which is used for adding all the elements of Collection c at the end of the list.

In this tutorial we will see another variant of addAll(int index, Collection c) method.Syntax of the method is :

public boolean addAll(int index, Collection c)

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



      // ArrayList1
      ArrayList<String> al = new ArrayList<String>();
      al.add("Apple");
      al.add("Orange");
      al.add("Grapes");
      al.add("Mango");
      System.out.println("ArrayList1 before addAll:"+al);
      //ArrayList2
      ArrayList<String> al2 = new ArrayList<String>();
      al2.add("Blackberry");
      al2.add("Strawberry");
      al2.add("Banana");
      al2.add("Guava");
      System.out.println("ArrayList2 content:"+al2);

      //Adding ArrayList2 in ArrayList1 at 4th position(index =3)
      al.addAll(3, al2);
      System.out.println("ArrayList1 after adding ArrayList2 at 4th Pos:\n"+al);
   }
}
```

Output :

```
ArrayList1 before addAll:[Apple, Orange, Grapes, Mango]
ArrayList2 content:[Blackberry, Strawberry, Banana, Guava]
ArrayList1 after adding ArrayList2 at 4th Pos:
[Apple, Orange, Grapes, Blackberry, Strawberry, Banana, Guava, Mango]
```

5.<u>Remove element from Specified Index in ArrayList</u>

In this tutorial we will learn about remove(int index) method which is used for removing an element at the specified index from an ArrayList. **Syntax:**

public Object remove(int index)

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

      //String ArrayList
      ArrayList<String> al = new ArrayList<String>();
      al.add("AA");
      al.add("BB");
      al.add("CC");
      al.add("DD");
      al.add("AA");
      al.add("ZZ");
      System.out.println("ArrayList before remove:");
      for(String var: al){
           System.out.println(var);
      }
      //Removing 1st element
      al.remove(0);
      //Removing 3rd element from the remaining list
      al.remove(2);
      //Removing 4th element from the remaining list
      al.remove(2);
      System.out.println("ArrayList After remove:");
      for(String var: al){
            System.out.println(var);
      }
   }
}
```

Output :

```
ArrayList before remove:
AA
BB
CC
DD
AA
ZZ
ArrayList After remove:
BB
CC
ZZ
```

6. Remove specified element from ArrayList

In this tutorial we will use remove(Object obj) method which removes the specified object from the list.

Syntax :

public boolean remove(Object obj)

According to Oracle docs, remove(Object obj) removes the first occurrence of the specified element from the list , if it is present. It returns false if the specified element doesn't exist in the list.

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

      //String ArrayList
      ArrayList<String> al = new ArrayList<String>();
      al.add("AA");
      al.add("BB");
      al.add("CC");
      al.add("DD");
      al.add("EE");
      al.add("FF");
      System.out.println("ArrayList before remove:");
      for(String var: al){
         System.out.println(var);
      }
      //Removing element AA from the arraylist
      al.remove("AA");
      //Removing element FF from the arraylist
      al.remove("FF");
      //Removing element CC from the arraylist
      al.remove("CC");
      /*This element is not present in the list so
       * it should return false
       */
      boolean bool=al.remove("GG");
      System.out.println("Element GG removed: "+bool);
      System.out.println("ArrayList After remove:");
      for(String var: al){
         System.out.println(var);
      }
   }
}
```

Output :

```
ArrayList before remove:
AA
BB
CC
DD
EE
FF
Element GG removed: false
ArrayList After remove:
BB
DD
EE
```

ArrayList Internal Working

Q.<u>How Add() method works Internally in ArrayList</u> ?

Before going into the details , first look at the code example of the ArrayList add(Object) method :

```
public class JavaHungry {

   public static void main(String[] args)
   {
      // TODO Auto-generated method stub

      ArrayList<Object> arrobj = new ArrayList<Object>();
      arrobj.add(3);
      arrobj.add("JavaCoffee");
      arrobj.add("4all");
      System.out.println(" is "+ arrobj);
   }
}
```

Output : [3, JavaCoffee, 4all]

So in the above example , we have created an ArrayList object arrobj . To add elements into the arrobj we called the add method on arrobj. After printing the arrobj , we get the desired result ,i.e , values are added to the arrobj.

But the question is how add(Object) method adds the value in ArrayList. So lets find out :

- There are two overloaded add() methods in ArrayList class:
  1. **add(Object) : adds object to the end of the list.**
  2. **add(int index , Object) : inserts the specified object at the specified position in the list.**

As internal working of both the add methods are almost similar. Here in this post , we will look in detail about the internal working of ArrayList add(Object) method.

- ArrayList **internally uses array object** to add(or store) the elements. In other words, ArrayList is backed by Array data -structure.The array of ArrayList is **resizable (or dynamic).**
- If you look into the ArrayList Api in jdk rt.jar , you will find the following code snippets in it.
  private transient Object[] elementData;

When you create the **ArrayList object** i.e ***new ArrayList()*** , the following code is executed :

this.elementData = new Object[initialCapacity];

There are two ways to create an ArrayList object .

a. Creates the empty list with initial capacity

1. List arrlstObj = new ArrayList();

When we create ArrayList this way , the default constructor of the ArrayList class is invoked. It will create internally an array of Object with default size set to 10.

2. List arrlstObj = new ArrayList(20);

When we create ArrayList this way , the ArrayList will invoke the constructor with the integer argument. It will create internally an array of Object . The size of the Object[] will be equal to the argument passed in the constructor . Thus when above line of code is executed ,it creates an Object[] of capacity 20.

Thus , above ArrayList constructors will create an empty list . Their initial capacity can be 10 or equal to the value of the argument passed in the constructor.

b. Creates the non empty list containing the elements of the specified collection.

List arrlstObj = new ArrayList(Collection c);

The above ArrayList constructor will create an non empty list containing the elements of the collection passed in the constructor.

How the size of ArrayList grows dynamically?

Inside the add(Object) , you will find the following code :

```
   public boolean add(E e)
   {
     ensureCapacity(size+1);
    elementData[size++] = e;
    return true;
}
```

- Important point to note from above code is that we are checking the capacity of the ArrayList , before adding the element.
- *ensureCapacity()* determines what is the current size of occupied elements and what is the maximum size of the array.

- If size of the filled elements (including the new element to be added to the ArrayList class) is greater than the maximum size of the array then increase the size of array.
- But the size of the array can not be increased dynamically. So what happens internally is new Array is created with capacity

- Till Java 6
  int newCapacity = (oldCapacity * 3)/2 + 1;

- (Update) From Java 7
  int newCapacity = oldCapacity + (oldCapacity >> 1);

- also, data from the old array is copied into the new array.

  Interviewer : Which copy technique internally used by the ArrayList class clone() method?
  There are two copy techniques present in the object oriented programming language , ***deep copy and shallow copy*.**

Just like HashSet , ArrayList also returns the shallow copy of the HashSet object. It means elements themselves are not cloned. In other words, shallow copy is made by copying the reference of the object.

Interviewer : How to create ArrayList ?
One liner answer :   List
Interviewer : What happens if ArrayList is concurrently modified while iterating the elements ?
According to ArrayList Oracle Java docs , The iterators returned by the ArrayList class's iterator and listiterator method are *fail-fast*.
Interviewer : What is the runtime performance of the get() method in ArrayList , where n represents the number of elements ?
get() ,set() , size() operations run in constant time i.e O(1)
add() operation runs in amortized constant time , i.e adding n elements require O(n) time.

```
import java.util.Arrays;
public class ArrayList<E> {


 private int size = 0;
  private static final int DEFAULT_CAPACITY = 10;
  private Object elements[];
  public ArrayList() {
     elements = new Object[DEFAULT_CAPACITY];
  }
public void add(E e) {
        if (size == elements.length) {
            ensureCapacity();
        }
        elements[size++] = e;
    }

private void ensureCapacity() {
        int newSize = elements.length * 2;
        elements = Arrays.copyOf(elements, newSize);
    }
    @SuppressWarnings("unchecked")
    public E get(int i) {
        if (i>= size || i <0) {
            throw new IndexOutOfBoundsException("Index: " + i + ", Size " + i );
        }
        return (E) elements[i];
    }
}

public class MyListTest {
  public static void main(String args[]){
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(3);
        list.add(4);
        System.out.println(4 == list.get(4));
        System.out.println (2 == list.get(1));
        System.out.println (3 == list.get(2));
       System.out.println(list.get(6));
    }
}
```

4(b). LinkedList class

Java LinkedList implements the List interface. It has the following properties:

1. LinkedList is a Doubly-linked list implementation of the List and Deque interfaces.
2. LinkedList can contain duplicate elements.
3. LinkedList class maintains insertion order.
4. LinkedList class manipulation is fast because no shifting needs to be occurred.
5. Java LinkedList class is unsynchronized.
6. Java LinkedList class can act as a stack, queue or list.

Doubly Linked List

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

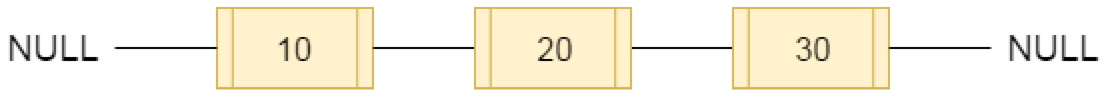


fig- doubly linked list

LinkedList Methods

1. **public boolean add(E e) :** it adds the specified element to the end of the list.
2. **public void addFirst(E e) :** it adds the specified element at the beginning of the list.
3. **public void addLast(E e) :** it adds the specified element at the end of the list.
4. **public void clear() :** it removes all of the elements from the list.
5. **public Object clone() :** it returns the shallow copy of the LinkedList.
6. **public boolean contains(Object o) :** it returns true if the list contains the specified element.
7. **public E get(int index) :** it returns the element in the specified position in the list.
8. **public E remove() :** it retrieves and remove the head of the list.
9. **public E remove(Object o) :** it removes the first occurrence of the specified element from the list if it is present.
10. **public E removeFirst() :** it removes and returns the first element of the list.
11. **public E removeLast() :** it removes and returns the last element of the list.
12. **public int size() :** it returns the number of elements in the list.

```
import java.util.*;

public class LinkedListExample {

  public static void main(String args[]) {

    // Declare LinkedList
    LinkedList<String> ll=new LinkedList<String>();

    // Adding elements to the LinkedList
      ll.add("A");
      ll.add("B");
      ll.addLast("C");
      ll.addFirst("D");
      ll.add(2, "E");
      ll.add("F");
      ll.add("G");
      System.out.println("LinkedList : " + ll);

      // Removing elements from the LinkedList
      ll.remove("C");
      ll.remove(2);
      ll.removeFirst();
      ll.removeLast();
      System.out.println("LinkedList after deletion: " + ll);

      // Finding elements in the LinkedList
      boolean status = ll.contains("A");

      if(status)
         System.out.println("List contains the element 'A' ");
      else
         System.out.println("List doesn't contain the element 'A'");

      // Number of elements in the LinkedList
      int size = ll.size();
      System.out.println("Size of LinkedList = " + size);

      // Get and set elements from LinkedList
      Object element = ll.get(2);
      System.out.println("Element returned by get() : " + element);
      ll.set(1, "Z");
      System.out.println("LinkedList after change : " + ll);
   }
}
```

Output :

```
LinkedList : [D, A, E, B, C, F, G]
LinkedList after deletion: [A, B, F]
List contains the element 'A'
Size of LinkedList = 3
Element returned by get() : F
LinkedList after change : [A, Z, F]
```

ArrayList vs LinkedList

Difference between ArrayList and LinkedList in Java

1. Implementation : ArrayList is the resizable array implementation of list interface , while LinkedList is the Doubly-linked list implementation of the list interface.

2. Performance : Performance of ArrayList and LinkedList depends on the type of operation

a. get(int index) or search operation : ArrayList get(int index) operation runs in constant time i.e O(1) while LinkedList get(int index) operation run time is O(n) .

The reason behind ArrayList being faster than LinkedList is that ArrayList uses index based system for its elements as it internally uses array data structure , on the other hand ,
LinkedList does not provide index based access for its elements as it iterates either from the beginning or end (whichever is closer) to retrieve the node at the specified element index.

b. insert() or add(Object) operation : Insertions in LinkedList are generally fast as compare to ArrayList.

In LinkedList adding or insertion is O(1) operation . While in ArrayList, if array is full i.e worst case, there is extra cost of resizing array and copying elements to the new array , which makes runtime of add operation in ArrayList O(n) , otherwise it is O(1) .

c. remove(int) operation : Remove operation in LinkedList is generally same as ArrayList i.e. O(n).

In LinkedList , there are two overloaded remove methods. one is remove() without any parameter which removes the head of the list and runs in constant time O(1) .
The other overloaded remove method in LinkedList is remove(int) or remove(Object) which removes the Object or int passed as parameter . This method traverses the LinkedList until it found the Object and unlink it from the original list . Hence this method run time is O(n).

While in ArrayList remove(int) method involves copying elements from old array to new updated array , hence its run time is O(n).

3. Reverse Iterator : LinkedList can be iterated in reverse direction using descendingIterator() while there is no descendingIterator() in ArrayList , so we need to write our own code to iterate over the ArrayList in reverse direction.

4. Initial Capacity : If the constructor is not overloaded , then ArrayList creates an empty list of initial capacity 10 , while LinkedList only constructs the empty list without any initial capacity.

5. Memory Overhead : Memory overhead in LinkedList is more as compared to ArrayList as node in LinkedList needs to maintain the addresses of next and previous node. While in ArrayList each index only holds the actual object(data).

Similarities between ArrayList and LinkedList :

1. Not synchronized : Both ArrayList and LinkedList are not synchronized , and can be made synchronized explicitly using Collections.synchronizedList() method.

2. clone() operation : Both ArrayList and LinkedList returns a shallow copy of the original object ,i.e. the elements themselves are not cloned.

3. Iterators : The iterators returned by ArrayList and LinkedList class's iterator and listIterator methods are fail-fast. Fail fast iterators throw ConcurrentModificationException .

4. Insertion Order : As ArrayList and LinkedList are the implementation of List interface,so, they both inherit properties of List . They both preserves the order of the elements in the way they are added to the ArrayList or LinkedList object.

	ArrayList	LinkedList
Implementation	Resizable Array	Douby-LinkedList
ReverseIterator	No	Yes , descendingIterator()
Initial Capacity	10	Constructs empty list
get(int) operation	Fast	Slow in comparision
add(int) operation	Slow in comparision	Fast
Memory Overhead	No	Yes

4(c). Vector class

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).

Commonly used methods of Vector Class:

1. **void addElement(Object element):** It inserts the element at the end of the Vector.
2. **int capacity():** This method returns the current capacity of the vector.
3. **int size():** It returns the current size of the vector.
4. **void setSize(int size):** It changes the existing size with the specified size.
5. **boolean contains(Object element):** This method checks whether the specified element is present in the Vector. If the element is been found it returns true else false.
6. **boolean containsAll(Collection c):** It returns true if all the elements of collection c are present in the Vector.
7. **Object elementAt(int index):** It returns the element present at the specified location in Vector.
8. **Object firstElement():** It is used for getting the first element of the vector.
9. **Object lastElement():** Returns the last element of the array.
10. **Object get(int index):** Returns the element at the specified index.

11. **boolean isEmpty():** This method returns true if Vector doesn’t have any element.
12. **boolean removeElement(Object element):** Removes the specifed element from vector.
13. **boolean removeAll(Collection c):** It Removes all those elements from vector which are present in the Collection c.
14. **void setElementAt(Object element, int index):** It updates the element of specifed index with the given element.

```
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() + " ");
   }
}
```

Output:

```
Size is: 4
Default capacity increment is: 4
Size after addition: 7
Capacity after increment is: 8
```

```
Elements are:
Apple Orange Mango Fig fruit1 fruit2 fruit3
```

Arraylist vs Vector in Java

1. Synchronization and Thread-Safe :
Vector is synchronized while ArrayList is not synchronized . Synchronization and thread safe means at a time only one thread can access the code .In Vector class all the methods are synchronized .Thats why the Vector object is already synchronized when it is created .

2. Performance :
Vector is slow as it is thread safe . In comparison ArrayList is fast as it is non synchronized . Thus in ArrayList two or more threads can access the code at the same time , while Vector is limited to one thread at a time.

3. Automatic Increase in Capacity :
A Vector defaults to doubling size of its array . While when you insert an element into the ArrayList , **it increases its Array size by 50% .**
By default ArrayList size is 10 . It checks whether it reaches the last element then it will create the new array ,copy the new data of last array to new array ,then old array is garbage collected by the Java Virtual Machine (JVM) .

4. Set Increment Size :
ArrayList does not define the increment size . Vector defines the increment size .
You can find the following method in Vector Class
public synchronized void setSize(int i) { //some code }
There is no setSize() method or any other method in ArrayList which can manually set the increment size.

5. Enumerator :
Other than Hashtable ,Vector is the only other class which uses both Enumeration and Iterator .While ArrayList can only use Iterator for traversing an ArrayList .

6. Introduction in Java :
java.util.Vector class was there in java since the very first version of the java development kit (jdk). java.util.ArrayList was introduced in java version 1.2 , as part of Java Collections framework . In java version 1.2 , Vector class has been refactored to implement the List Inteface .

5. Set Interface

- Set is an interface which extends Collection. It is an unordered collection of objects in which duplicate values cannot be stored (or) it makes no guarantee about the sequence of elements once you iterate them.
- Basically, Set is implemented by HashSet, LinkedSet or TreeSet (sorted representation).
- Set has various methods to add, remove clear, size, etc to enhance the usage of this interface.

```
// Java code for adding elements in Set
import java.util.*;
public class Set_example
{
   public static void main(String[] args)
   {
      // Set deonstration using HashSet
      Set<String> hash_Set = new HashSet<String>();
      hash_Set.add("Kaliber");
      hash_Set.add("Mind");
      hash_Set.add("Kaliber");
      hash_Set.add("Example");
      hash_Set.add("Set");
      System.out.print("Set output without the duplicates");
      System.out.println(hash_Set);

      // Set deonstration using TreeSet
      System.out.print("Sorted Set after passing into TreeSet");
      Set<String> tree_Set = new TreeSet<String>(hash_Set);
      System.out.println(tree_Set);
   }
}
```

(Please note that we have entered a duplicate entity but it is not displayed in the output. Also, we can directly sort the entries by passing the unordered Set in as the parameter of TreeSet).

Output:

```
Set output without the duplicates[Kaliber, Example, For, Set]

Sorted Set after passing into TreeSet[Example, For, Kaliber, Set]
```

Note: As we can see the duplicate entry “Kaliber” is ignored in the final output, Set interface doesn’t allow duplicate entries.

5(a). HashSet class

- Implements Set Interface.
- Underlying data structure for HashSet is **hashtable.**

- HashSet doesn't maintain any order, the elements would be returned in any random order.
- HashSet doesn't allow duplicates. If you try to add a duplicate element in HashSet, the old value would be overwritten.
- HashSet allows null values however if you insert more than one nulls it would still return only one null value.
- HashSet is non-synchronized. However it can be synchronized explicitly like this: **Set s = Collections.synchronizedSet(new HashSet(...));**
- The iterator returned by this class is fail-fast which means iterator would throw **ConcurrentModificationException** if HashSet has been modified after creation of iterator, by any means except iterator's own remove method.

Constructors in HashSet:

HashSet h = new HashSet();
Default initial capacity is 16 and default load factor is 0.75.

HashSet h = new HashSet(int initialCapacity);
default loadFactor of 0.75

HashSet h = new HashSet(int initialCapacity, float loadFactor);
HashSet h = new HashSet(Collection C);

What is initial capacity and load factor?

- The initial capacity means the number of buckets when hashtable (HashSet internally uses hashtable data structure) is created. Number of buckets will be automatically increased if the current size gets full.
- The load factor is a measure of how full the HashSet is allowed to get before its capacity is automatically increased.
- When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

```
                    Number of stored elements in the table

load factor = ----------------------------------------

                        Size of the hash table
```

E.g. If internal capacity is 16 and load factor is 0.75 then, number of buckets will automatically get increased when table has 12 elements in it.

HashSet Methods

1. boolean add(Element e) : it adds the element e to the list.
2. **public void clear() :** it removes all of the elements from the set.
3. public Object clone() : It returns the shallow copy of the HashSet instance.
4. public boolean contains(Object o) : it returns true if the set contains the specified element.
5. public boolean isEmpty() : This method returns true if the set contains no elements.
6. public boolean remove(Object o) : it removes the specified element from the set if it is present.
7. public int size() : it returns the number of elements in the set.

```
// Java program to demonstrate working of HashSet using Iterator Interface.
import java.util.*;
class Test
{
        public static void main(String[]args)
        {
                HashSet<String> h = new HashSet<String>();
                // adding into HashSet
                h.add("India");
                h.add("Australia");
                h.add("South Africa");
                h.add("India");// adding duplicate elements
                //Addition of null values
                h.add(null);
                h.add(null);
                // printing HashSet
                System.out.println(h);
                System.out.println("List contains India or not:" + h.contains("India"));
                // Removing an item
                h.remove("Australia");
                System.out.println("List after removing Australia:"+h);
                // Iterating over hash set items
                System.out.println("Iterating over list:");
                Iterator<String> i = h.iterator();
                while (i.hasNext())
                        System.out.println(i.next());
        }
}
```

Output :

```
[null, South Africa, Australia, India]

List contains India or not:true

List after removing Australia:[null, South Africa, India]

Iterating over list:

null

South Africa

India
```

// Java program to demonstrate working of HashSet using for each loop.

```
import java.util.*;

 public class HashSetIteratorExample {
   public static void main(String args[])
    {
   // Declaring a HashSet
   HashSet<String> hashset = new HashSet<String>();
   // Add elements to HashSet
   hashset.add("Pear");
   hashset.add("Apple");
   hashset.add("Orange");
   hashset.add("Papaya");
   hashset.add("Banana");

   System.out.println("HashSet contains :");
   // Using for each loop
   for(String str : hashset){
      System.out.println(str);
   }
  }
 }
```

Output :

```
HashSet contains :
Apple
Pear
Papaya
Orange
Banana
```

How HashSet works in Java

- **HashSet** uses HashMap internally to store it's objects. Whenever you create a HashSet object, one **HashMap** object associated with it is also created.
- This HashMap object is used to store the elements you enter in the HashSet.
- The elements you add into HashSet are stored as **keys** of this HashMap object. The value associated with those keys will be a **constant**.

Every constructor of HashSet class internally creates one HashMap object. You can check this in the source code of HashSet class in JDK installation directory. Below is the some sample code of the constructors of HashSet class.

```
private transient HashMap<E, object> map;


// Constructor - 1
// All the constructors are internally creating HashMap Object.
public HashSet()
{
    // Creating internally backing HashMap object
    map = new HashMap();
}
// Constructor - 2
public HashSet(int initialCapacity)
{
    // Creating internally backing HashMap object
    map = new HashMap< >(initialCapacity);
}
// Constructor - 3
public HashSet(int initialCapacity , float loadFactor)
{
  // Creating internally backing HashMap object
    map = new HashMap< >(initialCapacity, loadFactor);
}
```

You can notice that each and every constructor internally creates one new HashMap object.

```
public class JavaCoffee {

   public static void main(String[] args)
   {
      HashSet<Object> hashset = new HashSet<Object>();
      hashset.add(2);
      hashset.add("Java Coffee");
      hashset.add("Blogspot");
      hashset.add(2);                    // duplicate elements
      hashset.add("Java Coffee");              // duplicate elements
      System.out.println("Set is "+hashset);
   }
}
```

Output:

```
     Set is [3, Java Coffee, Blogspot]
```

Now , what happens internally when you pass duplicate elements in the add() method of the Set object , It will return false and do not add to the HashSet , as the element is already present .

But the main problem arises that how it returns false . So here is the answer.
When you open the HashSet implementation of the add() method in Java Apis that is rt.jar , you will find the following code in it :

```
public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, java.io.Serializable
{
   private transient HashMap<E,Object> map;
    // Dummy value to associate with an Object in the backing Map
    private static final Object PRESENT = new Object();

   public HashSet() {
      map = new HashMap<>();
   }

   // Let’s have a look at add() method of HashSet class.
   public boolean add(E e) {
      return map.put(e, PRESENT)==null;
   }
```

```
  //remove() method also works in the same manner. It internally calls remove method of Map
interface.
  public boolean remove(Object o) {
      return map.remove(o)==PRESENT;
   }
  //SOME CODES, i.e  Other methods in HashSet
}
```

- As we know in HashMap each key is unique . So what we do in the set is that we pass the argument in the add(Elemene E) that is E as a key in the HashMap .
- Now we need to associate some value to the key , so what Java apis developer did is to pass the Dummy value that is (new Object ()) which is referred by Object reference **PRESENT** . This **"PRESENT"** is defined in the HashSet class as below.

```
// Dummy value to associate with an Object in the backing Map
   private static final Object PRESENT = new Object();
```

- So , actually when you are adding a line in HashSet like hashset.add(2) . what java does internally is that it will put that element E here **2 as a key** in the HashMap(created during HashSet object creation) and some dummy value that is Object's object is passed as a value to the key .

```
public boolean add(E e)
{
        return map.put(e, PRESENT)==null;
}
```

You can notice that, add() method of HashSet class internally calls **put()** method of backing HashMap object by passing the element you have specified as a key and constant **"PRESENT"** as it's value.

- Now if you see the code of the HashMap put(Key k,Value V) method , you will find something like this:

```
 public V put(K key, V value) {
//Some code
}
```

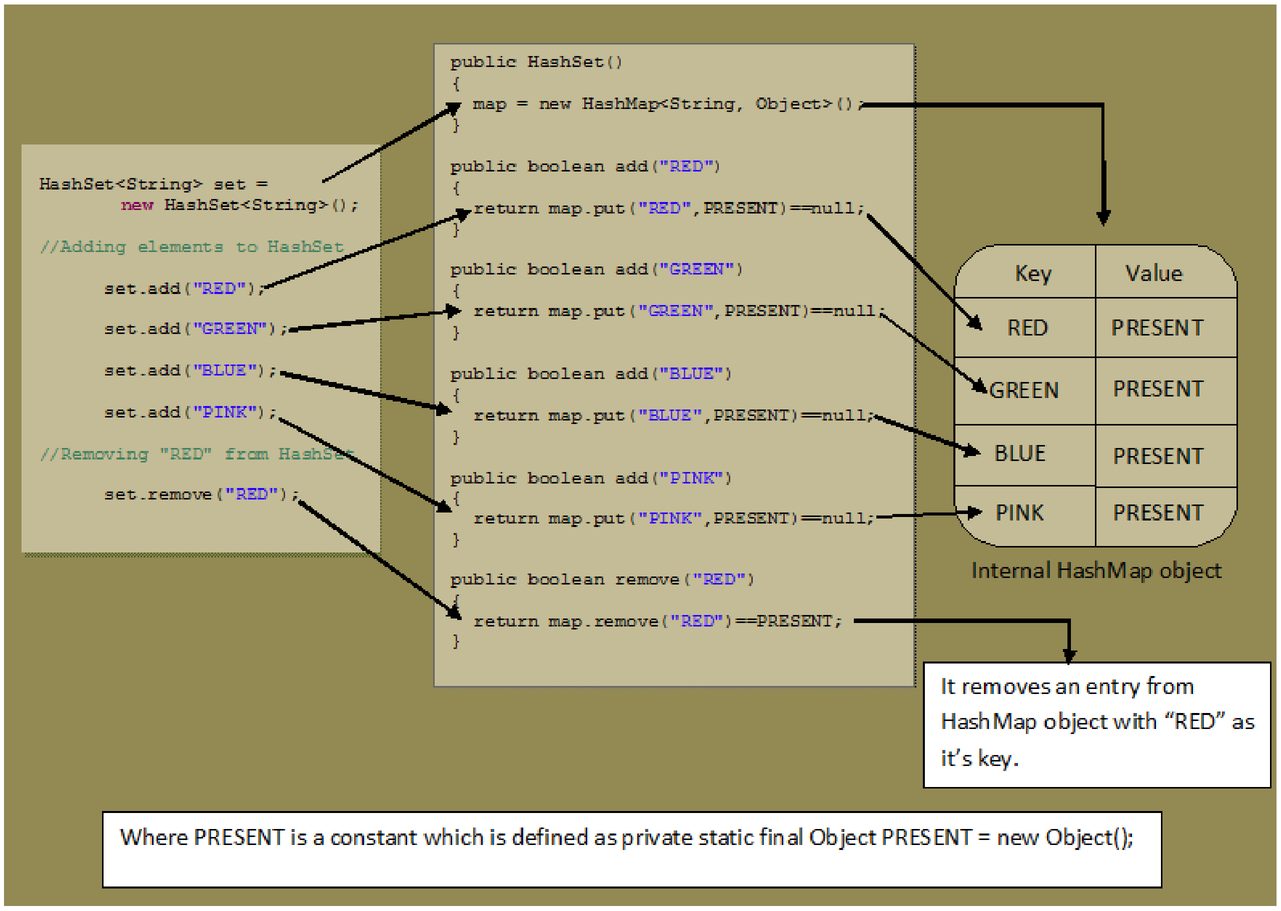
- The main point to notice in above code is that put (key,value) will return :
  1. null , if key is unique and added to the map.
  2. Old Value of the key , if key is duplicate.
- So , in HashSet add() method , we check the return value of map.put(key,value) method with null value.
  i.e.

```
public boolean add(E e) {
            return map.put(e, PRESENT)==null;
      }
```

So , if **map.put(key,value) returns null** ,then
map.put(e, PRESENT)==null   will return true and element is added to the HashSet.

So , if **map.put(key,value) returns old value of the key** ,then
map.put(e, PRESENT)==null   will return false and element is not added to the HashSet .

- See the below picture . You can observe that internal HashMap object contains elements of HashSet as keys and constant "PRESENT" as their value.



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

//Adding elements to HashSet

      set.add("RED");

      set.add("GREEN");

      set.add("BLUE");

      set.add("PINK");

//Removing "RED" from HashSet

      set.remove("RED");
```

```
public HashSet()
{
  map = new HashMap<String, Object>();
}

public boolean add("RED")
{
  return map.put("RED",PRESENT)==null;
}

public boolean add("GREEN")
{
  return map.put("GREEN",PRESENT)==null;
}

public boolean add("BLUE")
{
  return map.put("BLUE",PRESENT)==null;
}

public boolean add("PINK")
{
  return map.put("PINK",PRESENT)==null;
}

public boolean remove("RED")
{
  return map.remove("RED")==PRESENT;
}
```

Key	Value
RED	PRESENT
GREEN	PRESENT
BLUE	PRESENT
PINK	PRESENT

Internal HashMap object

It removes an entry from HashMap object with "RED" as it's key.

Where PRESENT is a constant which is defined as private static final Object PRESENT = new Object();

In the same manner, all methods of HashSet class process internally backing HashMap object to get the desired result. If you know how HashMap works, it will be easy for you to understand how HashSet works.

5(b). LinkedHashSet class

- A LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements.
- When the iteration order is needed to be maintained this class in used.
- When iterating through a HashSet the order is unpredictable, while a LinkedHashSet lets us iterate through the elements in the order in which they were inserted.
- when cycling through LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.

Syntax:

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

- Contains unique elements only like HashSet. It extends HashSet class and implements Set interface.
- Maintains insertion order.

```
import java.util.LinkedHashSet;
public class Demo
{
    public static void main(String[] args)
    {
        LinkedHashSet<String> linkedset =  new LinkedHashSet<String>();

        // Adding element to LinkedHashSet
        linkedset.add("A");
        linkedset.add("B");
        linkedset.add("C");
        linkedset.add("D");
       //This will not add new element as A already exists
        linkedset.add("A");
        linkedset.add("E");

        System.out.println("Size of LinkedHashSet = " + linkedset.size());
        System.out.println("Original LinkedHashSet:" + linkedset);
        System.out.println("Removing D from LinkedHashSet: " +
                        linkedset.remove("D"));
        System.out.println("Trying to Remove Z which is not "+
                        "present: " + linkedset.remove("Z"));
        System.out.println("Checking if A is present=" + linkedset.contains("A"));
        System.out.println("Updated LinkedHashSet: " + linkedset);
    }
}
```

Output:

```
Size of LinkedHashSet=5
Original LinkedHashSet:[A, B, C, D, E]
Removing D from LinkedHashSet: true
Trying to Remove Z which is not present: false
Checking if A is present=true
Updated LinkedHashSet: [A, B, C, E]
```

Observe the output: LinkedHashSet have preserved the insertion order.

How LinkedHashSet works Internally in Java

In this article, we will understand how HashSet subclass i.e LinkedHashSet works internally in java. Just like HashSet internally uses HashMap to add element to its object similarly LinkedHashSet internally uses LinkedHashMap to add element to its object .

Internal working of LinkedHashSet includes two basic questions ,first, How LinkedHashSet maintains Unique Elements ?, second , How LinkedHashSet maintains Insertion Order ? .

Why we need LinkedHashSet when we already have the HashSet and TreeSet ?

HashSet and TreeSet classes were added in jdk 1.2 while LinkedHashSet was added to the jdk in java version 1.4
HashSet provides constant time performance for basic operations like (add, remove and contains) method but **elements are in chaotic ordering i.e unordered.**
In TreeSet elements are naturally sorted but **there is increased cost associated with it .**
So , LinkedHashSet is added in jdk 1.4 to maintain ordering of the elements without incurring increased cost.

- **How LinkedHashSet Works Internally in Java ?**
  Before understanding how LinkedHashSet works internally in java in detail, we need to understand two terms *initial capacity* and *load factor* .

- **What is Initial capacity and load factor?**
  The ***capacity*** is the number of buckets(used to store key and value) in the Hash table , and the ***initial capacity*** is simply the capacity at the time Hash table is created.
  The ***load factor*** is a measure of how full the Hash table is allowed to get before its capacity is automatically increased.

- **Constructor of LinkedHashSet depends on above two parameters *initial capacity* and *load factor* .**
  There are **four constructors present in the LinkedHashSet class .**
  All constructors have the same below pattern :

```
// Constructor 1
public LinkedHashSet (int initialCapacity , float loadFactor)
{
   super(initialCapacity , loadFactor , true);
}
```

Note : If initialCapacity or loadFactor parameter value is missing during LinkedHashSet object creation , then default value of initialCapacity or loadFactor is used .
Default value for initialCapacity : **16** ,
Default value for loadFactor : **0.75f**

For example,
check the **below overloaded constructor , loadFactor is missing** in the LinkedHashSet constructor argument. So during super() call , we use the default value of the loadFactor(0.75f).

```
// Constructor
 public LinkedHashSet (int initialCapacity)
{
   super(initialCapacity , 0.75f , true);
}
```

check the **below overloaded constructor , initialCapacity and loadFactor both are missing** in the LinkedHashSet constructor argument. So during super() call , we use the default value of both initialCapacity(16) and loadFactor(0.75f).

```
// Constructor 3
 public LinkedHashSet ()
{
   super(16 , 0.75f , true);
}
```

below is the last overloaded constructor which uses Collection in the LinkedHashSet constructor argument. So during super() call , we use the default value of loadFactor(0.75f).

```
// Constructor 4
 public LinkedHashSet (Collection c)
{
   super(Math.max(2*c.size() ,11) , 0.75f , true);
}
```

Note : Since LinkedHashSet extends HashSet class.
Above all the 4 constructors are calling the super class (i.e HashSet) constructor , given below :

```
  public HashSet (int initialCapacity , float loadFactor , boolean dummy)
  {
            map = new LinkedHashMap<>(initialCapacity , loadFactor);
  }
```

In the above HashSet constructor , there are two main points to notice :
a. We are using extra boolean parameter ***dummy*** . It is used to differentiate this constructor from other(int, float) constructors of HashSet class which take initial capacity and load factor as their arguments.
b. Internally it is creating a LinkedHashMap object passing the initialCapacity and loadFactor as parameters.

As you are seeing, this constructor internally creates one new **LinkedHashMap** object. This LinkedHashMap object is used by the LinkedHashSet to store it's elements.

How LinkedHashSet Maintains Unique Elements ?

```
public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable,
java.io.Serializable
{
  private transient HashMap<E,Object> map;

  // Dummy value to associate with an Object in the backing Map
  private static final Object PRESENT = new Object();

  public HashSet(int initialCapacity , float loadFactor , boolean dummy) {

      map = new LinkedHashMap<>(initialCapacity , loadFactor);
  }
   // SOME CODE ,i.e Other methods in Hash Set

   public boolean add(E e) {
     return map.put(e, PRESENT)==null;
  }
  // SOME CODE ,i.e Other methods in Hash Set
}
```

- So , we are achieving uniqueness in LinkedHashSet,internally in java through LinkedHashMap . Whenever you create an object of LinkedHashSet it will indirectly create an object of LinkedHashMap as you can see in the italic lines of HashSet constructor.
- As we know in LinkedHashMap each key is unique . So what we do in the LinkedHashSet is that we pass the argument in the add(Elemene E) that is **E as a key** in the LinkedHashMap .Now we need to associate some value to the key , so what Java apis developer did is to pass the Dummy value that is (new Object ()) which is referred by Object reference **PRESENT** as below:

  ```
  // Dummy value to associate with an Object in the backing Map
     private static final Object PRESENT = new Object();
  ```
- So , actually when you are adding a line in LinkedHashSet like linkedhashset.add(5) . what java does internally is that it will put that element E here 5 as a key in the LinkedHashMap(created during LinkedHashSet object creation) and some dummy value that is Object's object is passed as a value to the key .
- Since LinkedHashMap put(Key k , Value v) method does not have its own implementation . LinkedHashMap put(Key k , Value v) method uses HashMap put(Key k , Value v) method.

- Now if you see the code of the HashMap put(Key k,Value v) method , you will find something like this :

  ```
  public V put(K key, V value) {
  //Some code
  }
  ```
- The main point to notice in above code is that put (key,value) will return ,
  1. null , if key is unique and added to the map.
  2. Old Value of the key , if key is duplicate.
- So , in LinkedHashSet add() method , we check the return value of map.put(key,value) method with null value
  i.e.

  ```
  public boolean add(E e) {
              return map.put(e, PRESENT)==null;
         }
  ```

  So , **if map.put(key,value) returns null** ,then
  map.put(e, PRESENT)==null will return true and element is added to the LinkedHashSet.

  So , **if map.put(key,value) returns old value of the key** ,then
  map.put(e, PRESENT)==null will return false and element is not added to the LinkedHashSet .

How LinkedHashSet Maintains Insertion Order ?

- LinkedHashSet differs from HashSet because it maintains the insertion order .
  According to **LinkedHashSet Oracle docs** ,

LinkedHashSet implementation differs from HashSet in that it maintains a doubly-linked list running through all of its entries

- LinkedHashSet internally uses LinkedHashMap to add elements to its object.

- **What is Entry object?**
  LinkedHashMap consists of a static inner class named as Entry . **Each object of Entry represents a key,value pair**. This Entry<K, V> class extends **HashMap.Entry** class.The **key K** in the Entry object is the **value** which needs to be added to the LinkedHashSet object. The **value V** in the Entry object is any **dummy object** called **PRESENT**.

- The insertion order of elements into LinkedHashMap are maintained by adding two new fields to this class. They are **before** and **after**. These two fields hold the references to previous and next elements. These two fields make LinkedHashMap to function as a doubly linked list.

(OR)

- Insertion Order of the LinkedHashMap is maintained by two Entry fields head and tail , which stores the head and tail of the doubly linked list.

```
  transient LinkedHashMap.Entry  head;
  transient LinkedHashMap.Entry  tail;
```

- For double linked list we need to maintain the previous and next Entry objects for each Entry object .
- Entry fields ***before*** and ***after*** are used **to store the references to the previous and next Entry objects** .

```
private static class Entry<K, V> extends HashMap.Entry {
// These fields comprise the doubly linked list used for iteration.
   Entry<K, V> before, after ;
   Entry( int hash , K key , V value , HashMap.Entry<K, V>  next ) {
      super(hash, key, value, next);
   }
}
```

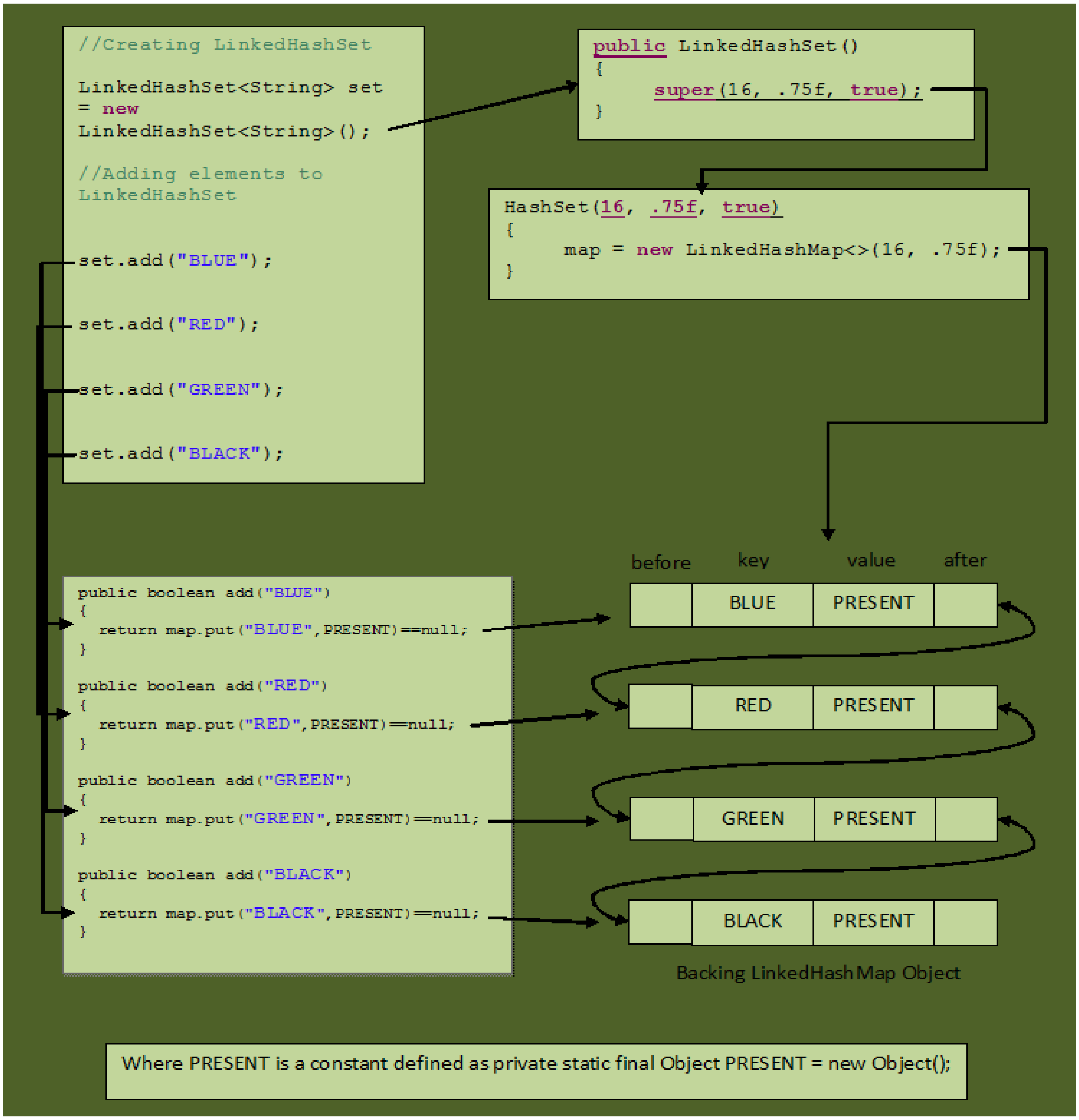
- The first two fields of above inner class of LinkedHashMap – **before** and **after** are responsible for maintaining the insertion order of the LinkedHashSet. The **header** field of LinkedHashMap stores the head of this doubly linked list. It is declared like below,

```
private transient Entry<K,V> header; //Stores the head of the doubly linked list
```

- Let’s see one example of LinkedHashSet to know how it works internally.

```
public class LinkedHashSetExample
{
   public static void main(String[] args)
   {
      //Creating LinkedHashSet
      LinkedHashSet<String> set = new LinkedHashSet<String>();
      //Adding elements to LinkedHashSet
      set.add("BLUE");
      set.add("RED");
      set.add("GREEN");
      set.add("BLACK");
    }
}
```

Look at the below image to see how above program works.



```
//Creating LinkedHashSet

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

//Adding elements to
LinkedHashSet

set.add("BLUE");

set.add("RED");

set.add("GREEN");

set.add("BLACK");
```

```
public LinkedHashSet()
{
     super(16, .75f, true);
}
```

```
HashSet(16, .75f, true)
{
     map = new LinkedHashMap<>(16, .75f);
}
```

```
public boolean add("BLUE")
{
  return map.put("BLUE",PRESENT)==null;
}

public boolean add("RED")
{
  return map.put("RED",PRESENT)==null;
}

public boolean add("GREEN")
{
  return map.put("GREEN",PRESENT)==null;
}

public boolean add("BLACK")
{
  return map.put("BLACK",PRESENT)==null;
}
```

Where PRESENT is a constant defined as private static final Object PRESENT = new Object();

If you know how LinkedHashMap works internally, it will be easy for you to understand how LinkedHashSet works internally. Go through source code of LinkedHashSet class and LinkedHashMap class once, you will get precise understanding about how LinkedHashSet works internally in Java.

SortedSet Interface in Java with Examples

SortedSet is an interface in collection framework. This interface extends Set and provides a total ordering of its elements. Exampled class that implements this interface is TreeSet.

```
              Collection              Map
       /     /    \      \            |
      /     /      \      \           |
    Set    List    Queue  Dequeue   SortedMap
    /
   /
SortedSet
           Core Interfaces in Collections
```

All elements of a SortedSet must implement the Comparable interface (or be accepted by the specified Comparator) and all such elements must be mutually comparable (i.e, Mutually Comparable simply means that two objects accept each other as the argument to their compareTo method)

Methods of Sorted Set interface:

1. **comparator() :** Returns the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements.
2. **first() :** Returns the first (lowest) element currently in this set.
3. **headSet(E toElement) :** Returns a view of the portion of this set whose elements are strictly less than toElement.
4. **last() :** Returns the last (highest) element currently in this set.
5. **subSet(E fromElement, E toElement) :** Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
6. **tailSet(E fromElement) :** Returns a view of the portion of this set whose elements are greater than or equal to fromElement.

```
public interface SortedSet extends Set
{
   // Range views
   SortedSet subSet(E fromElement, E toElement);
   SortedSet headSet(E toElement);
   SortedSet tailSet(E fromElement);
  // Endpoints
   E first();
   E last();
   // Comparator access
   Comparator comparator();
}
```

```
// A Java program to demonstrate working of SortedSet
import java.util.SortedSet;
import java.util.TreeSet;
public class Main
{
    public static void main(String[] args)
    {
        // Create a TreeSet and inserting elements
        SortedSet<String> sites = new TreeSet<>();
        sites.add("practice");
        sites.add("javaCoffee");
        sites.add("quiz");
        sites.add("code");

        System.out.println("Sorted Set: " + sites);
        System.out.println("First: " + sites.first());
        System.out.println("Last: " + sites.last());

        // Getting elements before quiz (Excluding) in a sortedSet
        SortedSet<String> beforeQuiz = sites.headSet("quiz");
        System.out.println(beforeQuiz);

        // Getting elements between code (Including) and // practice (Excluding)
        SortedSet<String> betweenCodeAndQuiz = sites.subSet("code","practice");
        System.out.println(betweenCodeAndQuiz);

        // Getting elements after code (Including)
        SortedSet<String> afterCode = sites.tailSet("code");
        System.out.println(afterCode);
    }
}
```

Output:

```
Sorted Set: [code, javaCoffee, practice, quiz]
First: code
Last: quiz
[code, javaCoffee, practice]
[code, javaCoffee]
[code, javaCoffee, practice, quiz]
```

5(C). TreeSet class

java.util.TreeSet is implementation class of SortedSet Interface. TreeSet has following important properties.

1. TreeSet implements the **SortedSet** interface so duplicate values are not allowed.
2. TreeSet does not preserve the insertion order of elements but elements are sorted by keys i.e. it sorts the element in ascending order.
3. TreeSet does not allow to insert Heterogeneous objects. It will throw **classCastException** at Runtime if trying to add hetrogeneous objects.
4. TreeSet doesn't allow to add null element.
5. TreeSet is basically implementation of a self-balancing binary search tree like **Red-Black Tree.** Therefore operations like add, remove and search take O(Log n) time. And operations like printing n elements in sorted order takes O(n) time.

Constructors:

Following are the four constructors in TreeSet class.

1. TreeSet t = new TreeSet();
This will create empty TreeSet object in which elements will get stored in default natural sorting order.

2. TreeSet t = new TreeSet(Comparator comp);
This constructor is used when you externally wants to specify sorting order of elements getting stored.

3. TreeSet t = new TreeSet(Collection col);
This constructor is used when we want to convert any Collection object to TreeSet object.

4. TreeSet t = new TreeSet(SortedSet s);
This constructor is used to convert SortedSet object to TreeSet Object.

Synchronized TreeSet:

Implementation of TreeSet class is not synchronized. If there is need of synchronized version of TreeSet, it can be done externally using Collections.synchronizedSet() method.

```
TreeSet ts = new TreeSet();
Set syncSet = Collections.synchronziedSet(ts);
```

Adding (or inserting) Elements to TreeSet:

TreeSet supports add() method to insert elements to it.

```
// Java program to demonstrate insertions in TreeSet
import java.util.*;

class TreeSetDemo
{
    public static void main (String[] args)
    {
        TreeSet ts1= new TreeSet();
        ts1.add("A");
        ts1.add("B");
        ts1.add("C");

        // Duplicates will not get insert
        ts1.add("C");
      // Elements get stored in default natural // Sorting Order(Ascending)
        System.out.println(ts1);  // [A,B,C]
        // ts1.add(2) ; will throw ClassCastException  at run time
    }
}
```

Output :

```
[A, B, C]
```

Null Insertion:

If we insert null in a TreeSet, it throws **NullPointerException** because while inserting null it will get compared to existing elements and null can not be compared to any value.

```
// Java program to demonstrate null insertion in TreeSet
import java.util.*;
 class TreeSetDemo
{
    public static void main (String[] args)
    {
        TreeSet ts2= new TreeSet();
        ts2.add("A");
        ts2.add("B");
        ts2.add("C");
        ts2.add(null); // Throws NullPointerException
    }
}
```

Output :

```
Exception in thread "main" java.lang.NullPointerException
    at java.util.TreeMap.put(TreeMap.java:563)
    at java.util.TreeSet.add(TreeSet.java:255)
    at TreeSetDemo.main(File.java:13)
```

Note: For empty tree-set, when you try to insert null as first value, you will get NPE from JDK 7.From 1.7 onwards null is not at all accepted by TreeSet. However upto JDK 6, null will be accepted as first value, but any if we insert any more value in TreeSet, it will also throw NullPointerException. Hence it was considered as bug and thus removed in JDK 7.

Methods:

TreeSet implements SortedSet so it has availability of all methods in Collection, Set and SortedSet interfaces. Following are the methods in Treeset interface.

1. **void add(Object o):** This method will add specified element according to some sorting order in TreeSet. Duplicate entires will not get added.
2. **boolean addAll(Collection c):** This method will add all elements of specified Collection to the set. Elements in Collection should be homogeneous otherwise ClassCastException will be thrown.Duplicate Entries of Collection will not be added to TreeSet.

```
// Java program to demonstrate TreeSet creation from
// ArrayList
import java.util.*;


class TreeSetDemo
{
    public static void main (String[] args)
    {
        ArrayList al = new ArrayList();
        al.add("JavaCoffee4all");
        al.add("JavaCoffee");
        al.add("Practice");
        al.add("Compiler");
        al.add("Compiler"); //will not be added
        // Creating a TreeSet object from ArrayList
        TreeSet ts4 = new TreeSet(al);

        // [Compiler,JavaCoffee,JavaCoffee4all,Practice]
        System.out.println(ts4);
    }
}
```

Output :

```
[Compiler, JavaCoffee, JavaCoffee4all, Practice]
```

3. **void clear() :** This method will remove all the elements.
4. **Comparator comparator():** This method will return Comparator used to sort elements in TreeSet or it will return null if default natural sorting order is used.
5. **boolean contains(Object o):** This method will return true if given element is present in TreeSet else it will return false.
6. **Object first() :** This method will return first element in TreeSet if TreeSet is not null else it will throw NoSuchElementException.
7. **Object last():** This method will return last element in TreeSet if TreeSet is not null else it will throw NoSuchElementException.
8. **SortedSet headSet(Object toElement):** This method will return elements of TreeSet which are less than the specified element.
9. **SortedSet tailSet(Object fromElement):** This method will return elements of TreeSet which are greater than or equal to the specified element.
10. **SortedSet subSet(Object fromElement, Object toElement):** This method will return elements ranging from fromElement to toElement. fromElement is inclusive and toElement is exclusive.

```
// Java program to demonstrate TreeSet creation from
// ArrayList
import java.util.*;
 class TreeSetDemo
{
    public static void main (String[] args)
    {
       TreeSet ts5 = new TreeSet();
        // Uncommenting below  throws NoSuchElementException
        // System.out.println(ts5.first());
        // Uncommenting below throws NoSuchElementException
        // System.out.println(ts5.last());
        ts5.add("JavaCoffee4all");
        ts5.add("Compiler");
        ts5.add("practice");

        System.out.println(ts5.first()); // Compiler
        System.out.println(ts5.last()); //Practice

        // Elements less than O. It prints
        // [Compiler,JavaCoffee4all]
        System.out.println(ts5.headSet("O"));

        // Elements greater than or equal to J.
        // It prints [JavaCoffee4all, Practice]
        System.out.println(ts5.tailSet("J"));
        // Elements ranging from C to P
        // It prints [Compiler,JavaCoffee4all]
        System.out.println(ts5.subSet("C","P"));
        // Deletes all elements from ts5.
        ts5.clear();
        // Prints nothing
        System.out.println(ts5);
    }
}
```

Output :

```
Compiler
practice
[Compiler, JavaCoffee4all]
[JavaCoffee4all, practice]
[Compiler, JavaCoffee4all]
[]
```

5(d). EnumSet class

EnumSet is one of the specialized implementation of Set interface for an enumeration type. It extends AbstractSet and implements Set Interface in Java.
It is a generic class declared as:

```
public abstract class EnumSet<E extends Enum<E>>
```

Here, E specifies the elements. E must extend Enum, which enforces the requirement that the elements must be of specified enum type.

Important:

- EnumSet class is a member of the Java Collections Framework & is not synchronized.
- It's a high performance set implementation, much faster than HashSet.
- All elements of each EnumSet instance must be elements of a single enum type.

Methods:

- **of(E e1, E e2) :** Creates an enum set initially containing the specified elements.
- **complementOf(EnumSet s) :** Creates an enum set with the same element type as the specified enum set, initially containing all the elements of this type that are not contained in the specified set.
- **allOf(Class elementType) :** Creates an enum set containing all of the elements in the specified element type.
- **range(E from, E to) :** Creates an enum set initially containing all of the elements in the range defined by the two specified endpoints.

```
// Java program to illustrate working of EnumSet and
// its functions.
import java.util.EnumSet;
enum Gfg
{
   CODE, LEARN, CONTRIBUTE, QUIZ, MCQ
};
public class Example
{
   public static void main(String[] args)
   {
      // create a set
      EnumSet<Gfg> set1, set2, set3, set4;
     // add elements
      set1 = EnumSet.of(Gfg.QUIZ, Gfg.CONTRIBUTE, Gfg.LEARN, Gfg.CODE);
      set2 = EnumSet.complementOf(set1);
```

```
        set3 = EnumSet.allOf(Gfg.class);
        set4 = EnumSet.range(Gfg.CODE, Gfg.CONTRIBUTE);
        System.out.println("Set 1: " + set1);
        System.out.println("Set 2: " + set2);
        System.out.println("Set 3: " + set3);
        System.out.println("Set 4: " + set4);
    }
}
```

Output:

```
Set 1: [CODE, LEARN, CONTRIBUTE, QUIZ]
Set 2: [MCQ]
Set 3: [CODE, LEARN, CONTRIBUTE, QUIZ, MCQ]
Set 4: [CODE, LEARN, CONTRIBUTE]
```

6. Map Interface

The java.util.Map interface represents a mapping between a key and a value. The Map interface is not a subtype of the Collection interface. Therefore it behaves a bit different from the rest of the collection types.

```
             Collection                Map
        /     /    \      \             |
       /     /      \      \            |
    Set    List    Queue  Dequeue   SortedMap
     /
    /
SortedSet
            Core Interfaces in Collections
```

- A Map cannot contain duplicate keys and each key can map to at most one value.
- Some implementations allow null key and null value (HashMap and LinkedHashMap) but some do not (TreeMap).
- The order of a map depends on specific implementations,
  e.g TreeMap and LinkedHashMap have predictable order, while HashMap does not.
  Exampled class that implements this interface is HashMap, TreeMap and LinkedHashMap.

Why and When Use Maps:

Maps are perfectly for key-value association mapping such as dictionaries. Use Maps when you want to retrieve and update elements by keys, or perform lookups by keys. Some examples:

- A map of error codes and their descriptions.
- A map of zip codes and cities.

- A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
- A map of classes and students. Each class (key) is associated with a list of students (value).

Methods of Map:

1. **public Object put(Object key, Object value) :-** is used to insert an entry in this map.
2. **public void putAll(Map map) :-** is used to insert the specified map in this map.
3. **public Object remove(Object key) :-** is used to delete an entry for the specified key.
4. **public Object get(Object key) :-** is used to return the value for the specified key.
5. **public boolean containsKey(Object key) :-** is used to search the specified key from this map.
6. **public Set keySet() :-** returns the Set view containing all the keys.
7. **public Set entrySet() :-** returns the Set view containing all the keys and values.

```
// Java program to demonstrate working of Map interface
import java.util.*;
class HashMapDemo
{
  public static void main(String args[])
  {
     HashMap< String,Integer> hm =
                      new HashMap< String,Integer>();
     hm.put("a", new Integer(100));
     hm.put("b", new Integer(200));
     hm.put("c", new Integer(300));
     hm.put("d", new Integer(400));
     // Returns Set view
     Set< Map.Entry< String,Integer> > st = hm.entrySet();

     for (Map.Entry< String,Integer> me:st)
     {
        System.out.print(me.getKey()+":");
        System.out.println(me.getValue());
     }
  }
}
```

Output:

```
a:100
b:200
```

```
c:300
d:400
```

6(a). HashMap class

- **HashMap** is a part of collection in Java since 1.2. It provides the basic implementation of Map interface of Java. It stores the data in (Key,Value) pairs.
- To access a value you must know its key, otherwise you can't access it. HashMap is known as HashMap because it uses a technique Hashing.
- **Definition of HashMap**

```
public class HashMap extends AbstractMap implements
Map, Cloneable, Serializable
```

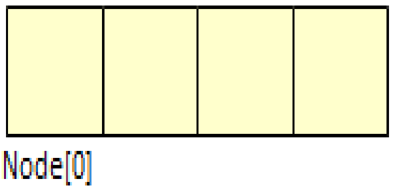
- HashMap *extends* AbstractMap *class and implements* Cloneable *and* Serializable *interfaces.*
- HashMap holds the data in the form of key-value pairs where each key is associated with one value.
- HashMap doesn't allow duplicate keys. But it can have duplicate values. That means A single key can't contain more than 1 value but more than 1 key can contain a single value.
- HashMap can have multiple null values and only one null key.
- HashMap maintains no order.
- HashMap gives constant time performance for the operations like *get()* and *put()* methods.
- Default initial capacity of *HashMap* is 16.

Internal Structure of HashMap

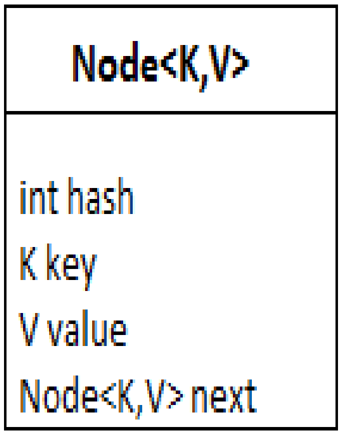
Internally HashMap contains an array of Node. and a node is represented as a class which contains 4 fields :

1. int hash
2. K key
3. V value
4. Node next

We can see that node is containing a reference of its own object. So it's a linked list. HashMap: :



Node:



How will you measure the performance of HashMap?

According to Oracle Java docs,
An instance of HashMap has two parameters that affect its performance: initial capacity and load factor.
The **capacity** is the number of buckets in the hash table(HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.), and the initial capacity is simply the capacity at the time the hash table is created.
The **load factor** is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.
In HashMap class, the default value of load factor is (.75) .

Synchronized HashMap

- As it is told that HashMap is unsynchronized i.e. multiple threads can access it simultaneously.
- If multiple threads access this class simultaneously and at least one thread manipulates it structurally then it is necessary to make it synchronized externally.
- It is done by synchronizing some object which enzapsulates the map. If No such object exists then it can be wrapped around **Collections.synchronizedMap()** to make HashMap synchronized and avoid accidental unsynchronized access. As in following example:

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

Now the Map m is synchronized.

- Iterators of this class are fail-fast if any structure modification is done after creation of iterator, in any way except through the iterator's remove method. In faliure of iterator it will throw **ConcurrentModificationException.**

Constructors of HashMap

HashMap provides 4 constructors and access modifier of each is public:

1. **HashMap() :** It is the default constructor which creates an instance of HashMap with initial capacity 16 and load factor 0.75.
2. **HashMap(int initial capacity) :** It creates a HashMap instance with specified initial capacity and load factor 0.75.
3. **HashMap(int initial capacity, float loadFactor) :** It creates a HashMap instance with specified initial capacity and specified load factor.
4. **HashMap(Map map) :** It creates instance of HashMapwith same mappings as specified map.

HashMap Methods:

1. **put():** java.util.HashMap.put() plays role in associating the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced.
   Syntax:

   ```
   public V put(K key,V value)
   ```

   Parameters:
   key - key with which the specified value is to be associated
   value - value to be associated with the specified key
   Return: the previous value associated with
   key, or null if there was no mapping for key.

2. **get():** java.util.HashMap.get()method returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
   Syntax:

   ```
   public V get(Object key)
   ```

   Parameters:
   key - the key whose associated value is to be returned
   Return: the value to which the specified
   key is mapped, or null if this map contains no mapping for
   the key.

3. **isEmpty():** java.util.HashMap.isEmpty() method returns true if the map contains no key-value mappings.
   Syntax:

   ```
   public boolean isEmpty()
   ```

   Return: true if this map contains no key-value mappings

4. **size():** java.util.HashMap.size() returns the number of key-value mappings in this map.
   Syntax:

   ```
   public int size()
   ```

   Return: the number of key-value mappings in this map.

Implementation to illustrate above methods

```
// Java program illustrating use of HashMap methods -
// put(), get(), isEmpty() and size()
import java.util.*;
public class NewClass
{
    public static void main(String args[])
    {
        // Creation of HashMap
        HashMap<String, String> hm = new HashMap<>();

        // Adding values to HashMap as ("keys", "values")
        hm.put("Language", "Java");
        hm.put("Platform", "JavaCoffee4all");
        hm.put("Code", "HashMap");
        hm.put("Learn", "More");

        System.out.println("Testing .isEmpty() method");

        // Checks whether the HashMap is empty or not
        // Not empty so printing the values
        if (!hm.isEmpty())
        {
            System.out.println("HashMap is notempty");

            // Accessing the contents of HashMap through Keys
            System.out.println("JavaCoffee : " + hm.get("Language"));
            System.out.println("JavaCoffee: " + hm.get("Platform"));
            System.out.println("JavaCoffee: " + hm.get("Code"));
            System.out.println("JavaCoffee: " + hm.get("Learn"));

            // size() method prints the size of HashMap.
            System.out.println("Size Of HashMap : " + hm.size());
        }
    }
}
```

Output

```
Testing .isEmpty() method

HashMap is notempty

JavaCoffee : Java

JavaCoffee : JavaCoffee4all
JavaCoffee : HashMap

JavaCoffee: More

Size Of HashMap : 4
```

5. keySet(): java.util.HashMap.keySet() It returns a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.

> **Syntax:**
> public Set keySet()
> **Return:** a set view of the keys contained in this map

6. values(): java.util.HashMap.values() It returns a Collection view of the values contained in this map. The collection is backed by the map, so changes to the map are reflected in the collection, and vice-versa.

> **Syntax:**
> public Collection values()
> **Return:** a collection view of the values contained in
> this map

7. containsKey(): java.util.HashMap.containsKey() It returns true if this map maps one or more keys to the specified value.

> **Syntax:**
> public boolean containsValue(Object value)
> **Parameters:**
> value - value whose presence in this map is to be tested
> **Return:** true if this map maps one or more keys to
> the specified value

Implementation:

```
// Java program illustrating usage of HashMap class methods
// keySet(), values(), containsKey()
import java.util.*;
public class NewClass
{
    public static void main(String args[])
    {
        // 1   Creation of HashMap
        HashMap<String, String> hm = new HashMap<>();

        // 2   Adding values to HashMap as ("keys", "values")
        hm.put("Language", "Java");
        hm.put("Platform", "JavaCoffee4all");
        hm.put("Code", "HashMap");
        hm.put("Learn", "More");
        // 3  containsKey() method is to check the presence
        //    of a particluar key
        // Since 'Code' key present here, the condition is true
        if (hm.containsKey("Code"))
           System.out.println("Testing .containsKey : " + hm.get("Code"));

        // 4 keySet() method returns all the keys in HashMap
        Set<String> hmkeys = hm.keySet();
        System.out.println("Initial keys  : " + hmkeys);


        // 5  values() method return all the values in HashMap
        Collection<String> hmvalues = hm.values();
        System.out.println("Initial values : " + hmvalues);

        // Adding new set of key-value
        hm.put("Search", "JavaArticle");

        // Again using .keySet() and .values() methods
        System.out.println("New Keys : " + hmkeys);
        System.out.println("New Values: " + hmvalues);
    }
}
```

Output:

```
Testing .containsKey : HashMap
Initial keys  : [Language, Platform, Learn, Code]
Initial values : [Java, JavaCoffee4all, More, HashMap]
New Keys : [Language, Platform, Search, Learn, Code]
New Values: [Java, JavaCoffee4all, JavaArticle, More, HashMap]
```

8. entrySet() : java.util.HashMap.entrySet() method returns a complete set of keys and values present in the HashMap.

> **Syntax:**
> public Set<Map.Entry> entrySet()
> **Return:**
> complete set of keys and values

9. getOrDefault : java.util.HashMap.getOrDefault() method returns a default value if there is no value find using the key we passed as an argument in HashMap. If the value for key if present already in the HashMap, it won't do anything to it.

It is very nice way to assign values to the keys that are not yet mapped, without interfering with the already present set of keys and values.

Syntax:
default V getOrDefault(Object key,V defaultValue)
Parameters:
key - the key whose mapped value we need to return
defaultValue - the default for the keys present in HashMap
Return:
mapping the unmapped keys with the default value.

10. **replace() : java.util.HashMap.replace(key, value)** or **java.util.HashMap.replace(key, oldvalue, newvalue)** method is a java.util.HashMap class method.
1st method accepts set of key and value which will replace the already present value of the key with the the new value passed in the argument. If no such set is present replace() method will do nothing.
Meanwhile 2nd method will only replace the already present set of key-old_value if the key and old_Value are found in the HashMap.

Syntax:
replace(k key, v value)
        or
replace(k key, v oldvalue, newvalue)
Parameters:
key   - key in set with the old value.
value   - new value we want to be with the specified key
oldvalue - old value in set with the specified key
newvalue - new value we want to be with the specified key
Return:
True - if the value is replaced
Null - if there is no such set present

11. **putIfAbsent java.util.HashMap.putIfAbsent(key, value)** method is being used to insert a new key-value set to the HashMap if the respective set is present. Null value is returned if such key-value set is already present in the HashMap.

Syntax:
public V putIfAbsent(key, value)
Parameters:
key   - key with which the specified value is associates.
value   - value to associates with the specified key.

```
// Java Program illustrating HashMap class methods().
// entrySet(), getOrDefault(), replace(), putIfAbsent
import java.util.*;
public class NewClass
{
   public static void main(String args[])
   {
      // Creation of HashMap
```

```
        HashMap<String, String> hm = new HashMap<>();

        // Adding values to HashMap as ("keys", "values")
        hm.put("Language", "Java");
        hm.put("Code", "HashMap");
        hm.put("Learn", "More");

        // .entrySet() returns all the keys with their values present in Hashmap
        Set<Map.Entry<String, String>> mappingHm = hm.entrySet();
        System.out.println("Set of Keys and Values using entrySet() : "+mappingHm);
        System.out.println();

        // Using .getOrDefault to access value
        // Here it is Showing Default value as key - "Code" was already present
        System.out.println("Using .getorDefault : "
                              + hm.getOrDefault("Code","javaArticle"));

        // Here it is Showing set value as key - "Search" was not present
        System.out.println("Using .getorDefault : "
                              + hm.getOrDefault("Search","javaArticle"));
        System.out.println();

        // .replace() method replacing value of key "Learn"
        hm.replace("Learn", "Methods");
        System.out.println("working of .replace()    : "+mappingHm);
        System.out.println();

        /* .putIfAbsent() method is placing a new key-value
           as they were not present initially*/
        hm.putIfAbsent("cool", "HashMap methods");
        System.out.println("working of .putIfAbsent() : "+mappingHm);

        /* .putIfAbsent() method is not doing anything
           as key-value were already present */
        hm.putIfAbsent("Code", "With_JAVA");
        System.out.println("working of .putIfAbsent() : "+mappingHm);

    }
}
```

Output:

```
Set of Keys and Values using entrySet() : [Language=Java, Learn=More, Code=HashMap]


Using .getorDefault : HashMap
Using .getorDefault : javaArticle


working of .replace()            : [Language=Java, Learn=Methods, Code=HashMap]


working of .putIfAbsent() : [Language=Java, cool=HashMap methods, Learn=Methods, Code=HashMap]

working of .putIfAbsent() : [Language=Java, cool=HashMap methods, Learn=Methods, Code=HashMap]
```

12. remove(Object key): Removes the mapping for this key from this map if present.

```
// Java Program illustrating remove() method using Iterator.
 import java.util.*;
public class NewClass
{
    public static void main(String args[])
    {
        //  Creation of HashMap
        HashMap<String, String> hm = new HashMap<>();

        //  Adding values to HashMap as ("keys", "values")
        hm.put("Language", "Java");
        hm.put("Platform", "JavaCoffee4all");
        hm.put("Code", "HashMap");


        //  .entrySet() returns all the keys with their values present in Hashmap
        Set<Map.Entry<String, String>> mappingHm = hm.entrySet();
        System.out.println("Set of Keys and Values : "+mappingHm);
        System.out.println();

        //  Creating an iterator
        System.out.println("Use of Iterator to remove the sets.");
        Iterator<Map.Entry<String, String>> hm_iterator = hm.entrySet().iterator();
        while(hm_iterator.hasNext())
        {
            Map.Entry<String, String> entry = hm_iterator.next();
            //  Removing a set one by one using iterator
            hm_iterator.remove(); // right way to remove entries from Map,
            // avoids ConcurrentModificationException
            System.out.println("Set of Keys and Values : "+mappingHm);

        }
    }
}
```

Output:

```
Set of Keys and Values : [Language=Java, Platform=JavaCoffee4all, Code=HashMap]


Use of Iterator to remove the sets.
Set of Keys and Values : [Platform=JavaCoffee4all, Code=HashMap]
Set of Keys and Values : [Code=HashMap]
Set of Keys and Values : []
```

6(b). LinkedHashMap class

HashMap in Java provides quick insert, search and delete operations. However it does not maintain any order on elements inserted into it. If we want to keep track of order of insertion, we can use LinkedHashMap.

LinkedHashMap is a Hash table and linked list implementation of the Map interface, with predictable iteration order. This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries.
LinkedHashMap is like HashMap with additional feature that we can access elements in their insertion order.

Syntax

```
LinkedHashMap<Integer, String> lhm = new LinkedHashMap<Integer, String>();
```

- A LinkedHashMap contains values based on the key. It implements the Map interface and extends HashMap class.
- It contains only unique elements .
- It may have one null key and multiple null values.
- It is same as HashMap with additional feature that it maintains insertion order. For example, when we ran the code with HashMap, we got different oder of elements .

```
import java.util.LinkedHashMap;
import java.util.Set;
import java.util.Iterator;
import java.util.Map;
public class LinkedHashMapDemo {
   public static void main(String args[]) {
       // LinkedHashMap Declaration
       LinkedHashMap<Integer, String> lhmap =
             new LinkedHashMap<Integer, String>();

       //Adding elements to LinkedHashMap
       lhmap.put(22, "Abey");
       lhmap.put(33, "Dawn");
```

```
        lhmap.put(1, "Sherry");
        lhmap.put(2, "Karon");
        lhmap.put(100, "Jim");

        // Generating a Set of entries
        Set set = lhmap.entrySet();

        // Displaying elements of LinkedHashMap
        Iterator iterator = set.iterator();
        while(iterator.hasNext()) {
          Map.Entry me = (Map.Entry)iterator.next();
          System.out.print("Key is: "+ me.getKey() +
                 "& Value is: "+me.getValue()+"\n");
        }
    }
}
```

Output:

```
Key is: 22& Value is: Abey
Key is: 33& Value is: Dawn
Key is: 1& Value is: Sherry
Key is: 2& Value is: Karon
Key is: 100& Value is: Jim
```

As you can see the values are returned in the same order in which they got inserted.

```
// Java program to demonstrate working of LinkedHashMap
import java.util.*;

public class BasicLinkedHashMap
{
   public static void main(String a[])
   {
      LinkedHashMap<String, String> lhm =
                  new LinkedHashMap<String, String>();
      lhm.put("one", "practice);
      lhm.put("two", "code");
      lhm.put("four", "quiz");

      // It prints the elements in same order as they were inserted
      System.out.println(lhm);

      System.out.println("Getting value for key 'one': " + lhm.get("one"));
      System.out.println("Size of the map: " + lhm.size());
      System.out.println("Is map empty? " + lhm.isEmpty());
      System.out.println("Contains key 'two'? "+ lhm.containsKey("two"));
```

```
        System.out.println("Contains value 'practice'? "
                        + lhm.containsValue("practice"));
        System.out.println("delete element 'one': " + lhm.remove("one"));
        System.out.println(lhm);
    }
}
```

Output:

```
{one=practice, two=code, four=quiz}

Getting value for key 'one': practice

Size of the map: 3

Is map empty? false

Contains key 'two'? true

Contains value 'practice'? true

delete element 'one': practice

{two=code, four=quiz}
```

6(c). TreeMap class

TreeMap class implements the Map interface.It extends AbstractMap class and implements NavigableMap interface. It has the following properties:

1. TreeMap can be a bit handy when we only need to store unique elements in a sorted order. i.e. TreeMap is sorted in the ascending order of its keys.

2. Java.util.TreeMap uses a red-black tree in the background which makes sure that there are no duplicates; additionally it also maintains the elements in a sorted order.

```
 TreeMap<K, V> hmap = new TreeMap<K, V>();
```

3. For operations like add, remove, containsKey, time complexity is O(log n where n is number of elements present in TreeMap.
4. TreeMap does not allow null keys and null values.
5. TreeMap is not synchronized.It must be synchronized externally
   SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));

Methods:

Method	Description
boolean containsKey(Object key)	It is used to return true if this map contains a mapping for the specified key.
boolean containsValue(Object value)	It is used to return true if this map maps one or more keys to the specified value.
Object firstKey()	It is used to return the first (lowest) key currently in this sorted map.
Object get(Object key)	It is used to return the value to which this map maps the specified key.
Object lastKey()	It is used to return the last (highest) key currently in this sorted map.
Object remove(Object key)	It is used to remove the mapping for this key from this TreeMap if present.
void putAll(Map map)	It is used to copy all of the mappings from the specified map to this map.
Set entrySet()	It is used to return a set view of the mappings contained in this map.
int size()	It is used to return the number of key-value mappings in this map.
Collection values()	It is used to return a collection view of the values contained in this map.

```
import java.util.TreeMap;
import java.util.Set;
import java.util.Iterator;
import java.util.Map;

public class TreeMapDemo {

  public static void main(String args[]) {

    /* This is how to declare TreeMap */
    TreeMap<Integer, String> tmap =  new TreeMap<Integer, String>();
```

```
      /*Adding elements to TreeMap*/
      tmap.put(1, "Data1");
      tmap.put(23, "Data2");
      tmap.put(70, "Data3");
      tmap.put(4, "Data4");
      tmap.put(2, "Data5");

      /* Display content using Iterator*/
      Set set = tmap.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());
      }

   }
}
```

Output:

```
key is: 1 & Value is: Data1
key is: 2 & Value is: Data5
key is: 4 & Value is: Data4
key is: 23 & Value is: Data2
key is: 70 & Value is: Data3
```

As you can see that we have inserted the data in random order however when we displayed the TreeMap content we got the sorted result in the ascending order of keys.

```
import java.util.*;
class TreeMapDemo2{
 public static void main(String args[]){

  TreeMap<Integer,String> hm=new TreeMap<Integer,String>();

  hm.put(100,"Amit");
  hm.put(102,"Ravi");
  hm.put(101,"Vijay");
  hm.put(103,"Rahul");

  for(Map.Entry m:hm.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }
  System.out.println("FirstKey :" +hm.firstKey());
  System.out.println("LastKey :" +hm.lastKey());
 }
}
```

Output:

```
100 Amit
101 Vijay
102 Ravi
103 Rahul
FirstKey :100
LastKey :103
```

6(d). EnumMap class

EnumMap is specialized implementation of Map interface for enumeration types.
It extends AbstractMap and implements Map Interface in Java. It is a generic class declared as:
Syntax:

```
public class EnumMap<K extends Enum<K>,V>
K: specifies the keys
V: specifies values
```

K must extend Enum, which enforces the requirement that the keys must be of specified enum type.
Important points:

- EnumMap class is a member of the Java Collections Framework & is not synchronized.
- EnumMap is ordered collection and they are maintained in the natural order of their keys(natural order of keys means the order on which enum constant are declared inside enum type)
- It's a high performance map implementation, much faster than HashMap.
- All keys of each EnumMap instance must be keys of a single enum type.
- EnumMap doesn't allow null key and throw NullPointerException, at same time null values are permitted.

```
// Java program to illustrate working of EnumMap and
// its functions.
import java.util.EnumMap;

public class Example
{
   public enum GFG
   {
      CODE, CONTRIBUTE, QUIZ, MCQ;
   }
```

```
    public static void main(String args[])
    {
        // Java EnumMap Example 1: creating EnumMap in java with key
        //as enum type STATE
        EnumMap<GFG, String> gfgMap = new EnumMap<GFG, String>(GFG.class);

        // Java EnumMap Example 2:
        // putting values inside EnumMap in Java
        // we are inserting Enum keys on different order than their natural order
        gfgMap.put(GFG.CODE, "Start Coding with gfg");
        gfgMap.put(GFG.CONTRIBUTE, "Contribute for others");
        gfgMap.put(GFG.QUIZ, "Practice Quizes");
        gfgMap.put(GFG.MCQ, "Test Speed with Mcqs");

        // printing size of EnumMap in java
        System.out.println("Size of EnumMap in java: " + gfgMap.size());

        // printing Java EnumMap , should print EnumMap in natural order
        // of enum keys (order on which they are declared)
        System.out.println("EnumMap: " + gfgMap);

        // retrieving value from EnumMap in java
        System.out.println("Key : " + GFG.CODE +" Value: "
                        + gfgMap.get(GFG.CODE));

        // checking if EnumMap contains a particular key
        System.out.println("Does gfgMap has :" + GFG.CONTRIBUTE + " : "
                        +  gfgMap.containsKey(GFG.CONTRIBUTE));

        // checking if EnumMap contains a particular value
        System.out.println("Does gfgMap has :" + GFG.QUIZ + " : "
                        + gfgMap.containsValue("Practice Quizes"));
        System.out.println("Does gfgMap has :" + GFG.QUIZ + " : "
                        + gfgMap.containsValue(null));
    }
}
```

Output:

```
Size of EnumMap in java: 4

EnumMap: {CODE=Start Coding with gfg, CONTRIBUTE=Contribute for others, QUIZ=Practice
Quizes

,MCQ=Test Speed with Mcqs}

Key : CODE Value: Start Coding with gfg
```

```
Does gfgMap has :CONTRIBUTE : true
Does gfgMap has :QUIZ : true
Does gfgMap has :QUIZ : false
```

6(e). HashTable class

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

The important points about Java Hashtable class are:

- A Hashtable is an array of list. Each list is known as a bucket. The position of bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
- It contains only unique elements.
- It may have not have any null key or value.
- It is synchronized.
- Hashtable doesn't preserve the insertion order, neither it sorts the inserted data based on keys or values. Which means no matter what keys & values you insert into Hashtable, the result would not be in any particular order.

Constructors:

- **Hashtable():** This is the default constructor.
- **Hashtable(int size):** This creates a hash table that has initial size specified by size.
- **Hashtable(int size, float fillRatio):** This version creates a hash table that has initial size specified by size and fill ratio specified by fillRatio. **fill ratio:** Basically it determines how full hash table can be before it is resized upward.and its Value lie between 0.0 to 1.0
- **Hashtable(Map m):** This creates a hash table that is initialised with the elements in m.

Methods of Java Hashtable class

Method	**Description**
void clear()	It is used to reset the hash table.
boolean contains(Object value)	This method return true if some value equal to the value exist within the hash table, else return false.

boolean containsValue(Object value)	This method return true if some value equal to the value exists within the hash table, else return false.
boolean containsKey(Object key)	This method return true if some key equal to the key exists within the hash table, else return false.
boolean isEmpty()	This method return true if the hash table is empty; returns false if it contains at least one key.
void rehash()	It is used to increase the size of the hash table and rehashes all of its keys.
Object get(Object key)	This method return the object that contains the value associated with the key.
Object remove(Object key)	It is used to remove the key and its value. This method return the value associated with the key.
int size()	This method return the number of entries in the hash table.

```
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 HashtableExample {
public static void main(String[] args) {
    //Creating map of Books
    Map<Integer,Book> map=new Hashtable<Integer,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 map
    map.put(1,b1);
    map.put(2,b2);
    map.put(3,b3);
    //Traversing map
    for(Map.Entry<Integer, Book> entry:map.entrySet()){
        int key=entry.getKey();
        Book b=entry.getValue();
        System.out.println(key+" Details:");
        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}
```

Output:

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

7. Queue Interface

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 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();
```

<u>Methods of Queue interface</u>

boolean add(E e): This method adds the specified element at the end of Queue. Returns true if the the element is added successfully or false if the element is not added that basically happens when the Queue is at its max capacity and cannot take any more elements.

E element(): This method returns the head (the first element) of the Queue.

boolean offer(object): This is same as add() method.

E remove(): This method removes the head(first element) of the Queue and returns its value.

E poll(): This method is almost same as remove() method. The only difference between poll() and remove() is that poll() method returns null if the Queue is empty.

E peek(): This method is almost same as element() method. The only difference between peek() and element() is that peek() method returns null if the Queue is empty.

```
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);
  }
}
```

Output:

```
Elements in Queue:[Rick, Maggie, Glenn, Negan, Daryl]
Removed element: Rick
Head: Maggie
poll(): Maggie
peek(): Glenn
Elements in Queue:[Glenn, Negan, Daryl]
```

7(a). PriorityQueue class

The **PriorityQueue** is a queue in which elements are ordered according to specified Comparator. You have to specify this Comparator while creating a PriorityQueue itself.

If no Comparator is specified, elements will be placed in their natural order. The PriorityQueue is a special type of queue because it is not a **First-In-First-Out** (FIFO) as in the normal queues. But, elements are placed according to supplied Comaparator.

The PriorityQueue does not allow **null** elements. Elements in the PriorityQueue must be of **Comparable** type, If you insert the elements which are not Comparable, you will get **ClassCastException** at run time.

PriorityQueue class extends **AbstractQueue** class which in turn implements **Queue** interface.

- Elements in the PriorityQueue are ordered according to supplied **Comparator**. If Comparator is not supplied, elements will be placed in their natural order.
- The PriorityQueue is **unbounded**. That means the capacity of the PriorityQueue increases automatically if the size exceeds capacity. But, how it grows is not specified.
- The PriorityQueue can have **duplicate** elements but can not have **null** elements.

- All elements of the PriorityQueue must be of **Comparable type**. Otherwise ClassCastException will be thrown at run time.
- The head element of the PriorityQueue is always the least element and tail element is always the largest element according to specified Comparator.
- The default initial capacity of PriorityQueue is **11**.
- You can retrieve the Comparator used to order the elements of the PriorityQueue using **comparator()** method.
- PriorityQueue is not a thread safe.

Constructor: PriorityQueue()
This creates a PriorityQueue with the default initial capacity that orders its elements according to their natural ordering.

Methods:

1. **boolean add(E element):** This method inserts the specified element into this priority queue.
2. **public remove():** This method removes a single instance of the specified element from this queue, if it is present
3. **public poll():** This method retrieves and removes the head of this queue, or returns null if this queue is empty.
4. **public peek():** This method retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
5. **iterator():** Returns an iterator over the elements in this queue.
6. **booleancontains(Object o):** This method returns true if this queue contains the specified element

```
// Java progrm to demonstrate working of priority queue in Java
import java.util.*;

class PriorityQueueDemo
{
   public static void main(String args[])
   {
      // Creating empty priority queue
      PriorityQueue<String> pQueue = new PriorityQueue<String>();

      // Adding items to the pQueue
      pQueue.add("C");
      pQueue.add("C++");
      pQueue.add("Java");
      pQueue.add("Python");
     // Printing the most priority element
      System.out.println("Head value using peek function:" + pQueue.peek());

      // Printing all elements
      System.out.println("The queue elements:");
      Iterator itr = pQueue.iterator();
```

```
        while (itr.hasNext())
           System.out.println(itr.next());

        // Removing the top priority element (or head) and
        // printing the modified pQueue
        pQueue.poll();
        System.out.println("After removing an element" + "with poll function:");
        Iterator<String> itr2 = pQueue.iterator();
        while (itr2.hasNext())
           System.out.println(itr2.next());

        // Removing Java
        pQueue.remove("Java");
        System.out.println("after removing Java with" + " remove function:");
        Iterator<String> itr3 = pQueue.iterator();
        while (itr3.hasNext())
           System.out.println(itr3.next());

        // Check if an element is present
        boolean b = pQueue.contains("C");
        System.out.println ( "Priority queue contains C" + "or not?: " + b);

        // get objects from the queue in an array and
        // print the array
        Object[] arr = pQueue.toArray();
        System.out.println ( "Value in array: ");
        for (int i = 0; i<arr.length; i++)
          System.out.println ( "Value: " + arr[i].toString()) ;
    }
}
```

Output:

```
Head value using peek function:C

The queue elements:

C

C++

Java

Python

After removing an elementwith poll function:

C++

Python
```

```
Java
after removing Java with remove function:
C++
Python
Priority queue contains C or not?: false
Value in array:
Value: C++
Value: Python
```

Java PriorityQueue Example With Customized Comparator :

In this example, we create a PriorityQueue with our own Comparator. We try to create a PriorityQueue of '**Employee**' objects ordered in the ascending order of their salaries. That means head element always will be an '**Employee**' object with lowest salary.

```
//Let's define 'Employee' class with two attributes –  'name' and 'salary'.
class Employee
{
    String name;
    int salary;
  //Constructor Of Employee
 public Employee(String name, int salary)
    {
        this.name = name;
        this.salary = salary;
    }
 //Here, toString() method is overrided so that it returns the contents of the object.
    @Override
    public String toString()
    {
        return name+" : "+salary;
    }
}

//Let's define our own Comparator class 'MyComparator' which compares the salary of two Employees.
class MyComparator implements Comparator<Employee>
{
    @Override
    public int compare(Employee e1, Employee e2)
    {
        return e1.salary - e2.salary;
    }
}
 // Let's create a PriorityQueue of 'Employee' objects with 'MyComparator' as a Comparator.
public class PriorityQueueExample
{
   public static void main(String[] args)
   {
      //Instantiating MyComaparator

      MyComparator comparator = new MyComparator();

      //Creating PriorityQueue of Employee objects with MyComparator as Comparator

      PriorityQueue<Employee> pQueue = new PriorityQueue<Employee>(7, comparator);

      //Adding Employee objects to pQueue

      pQueue.offer(new Employee("AAA", 15000));
      pQueue.offer(new Employee("BBB", 12000));
      pQueue.offer(new Employee("CCC", 7500));
      pQueue.offer(new Employee("DDD", 17500));
      pQueue.offer(new Employee("EEE", 21500));
      pQueue.offer(new Employee("FFF", 29000));
      pQueue.offer(new Employee("GGG", 14300));

      //Removing the head elements
      //remove the head elements of the 'pQueue' one by one.
      System.out.println(pQueue.poll());      //Output --> CCC : 7500
      System.out.println(pQueue.poll());      //Output --> BBB : 12000
      System.out.println(pQueue.poll());      //Output --> GGG : 14300
      System.out.println(pQueue.poll());      //Output --> AAA : 15000
      System.out.println(pQueue.poll());      //Output --> DDD : 17500
      System.out.println(pQueue.poll());      //Output --> EEE : 21500
      System.out.println(pQueue.poll());      //Output --> FFF : 29000
   }
}
```

You can notice that always an **Employee** of lowest salary is removed. That means, head element always contains **Employee** object with lowest salary. '**Employee**' objects in '**pQueue**' are placed in the ascending order of their salary.

8. Deque Interface

The java.util.Deque interface is a subtype of the java.util.Queue interface. The Deque is related to the double-ended queue that supports addition or removal of elements from either end of the data structure, it can be used as a queue (first-in-first-out/FIFO) or as a stack (last-in-first-out/LIFO).

Methods of deque:

1. **add(element):** Adds an element to the tail.
2. **addFirst(element):** Adds an element to the head.
3. **addLast(element):** Adds an element to the tail.
4. **offer(element):** Adds an element to the tail and returns a boolean to explain if the insertion was successful.
5. **offerFirst(element):** Adds an element to the head and returns a boolean to explain if the insertion was successful.
6. **offerLast(element):** Adds an element to the tail and returns a boolean to explain if the insertion was successful.
7. **iterator():** Returna an iterator for this deque.
8. **descendingIterator():** Returns an iterator that has the reverse order for this deque.
9. **push(element):** Adds an element to the head.
10. **pop(element):** Removes an element from the head and returns it.
11. **removeFirst():** Removes the element at the head.
12. **removeLast():** Removes the element at the tail.

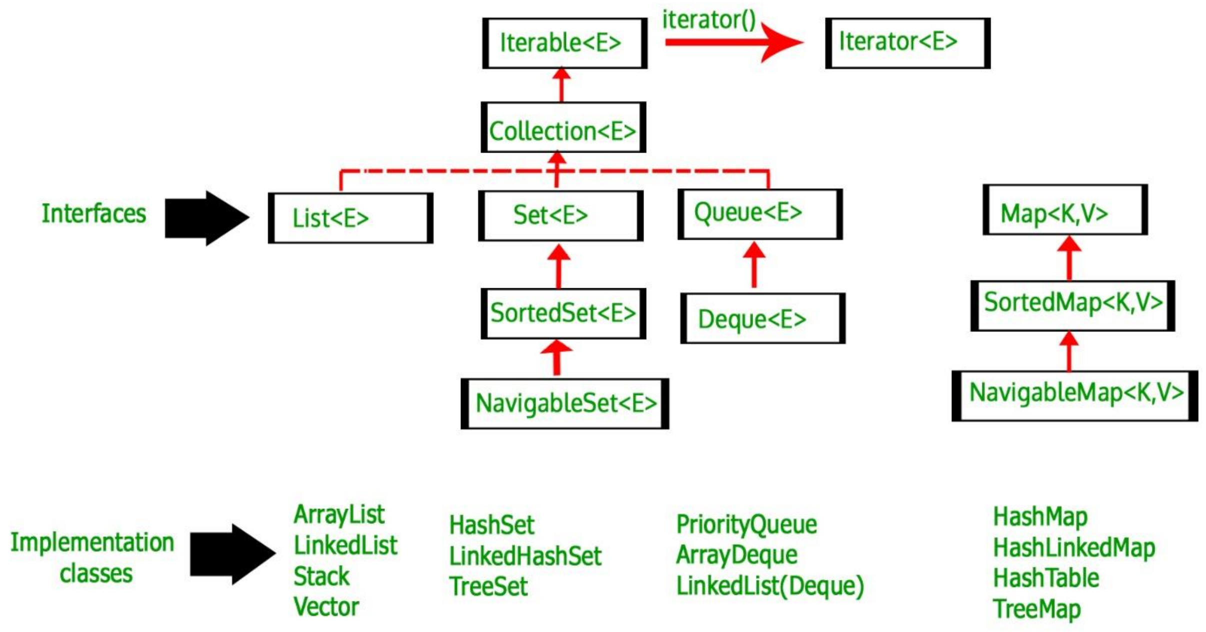
```
// Java program to demonstrate working of
// Deque in Java
import java.util.*;

public class DequeExample
{
   public static void main(String[] args)
   {
      Deque deque = new LinkedList<>();

      // We can add elements to the queue in various ways
      deque.add("Element 1 (Tail)"); // add to tail
      deque.addFirst("Element 2 (Head)");
      deque.addLast("Element 3 (Tail)");
      deque.push("Element 4 (Head)"); //add to head
      deque.offer("Element 5 (Tail)");
      deque.offerFirst("Element 6 (Head)");
      deque.offerLast("Element 7 (Tail)");
```

```
        System.out.println(deque + "\n");
       // Iterate through the queue elements.
        System.out.println("Standard Iterator");
        Iterator iterator = deque.iterator();
        while (iterator.hasNext())
            System.out.println("\t" + iterator.next());
       // Reverse order iterator
        Iterator reverse = deque.descendingIterator();
        System.out.println("Reverse Iterator");
        while (reverse.hasNext())
            System.out.println("\t" + reverse.next());

        // Peek returns the head, without deleting
        // it from the deque
        System.out.println("Peek " + deque.peek());
        System.out.println("After peek: " + deque);
        // Pop returns the head, and removes it from
        // the deque
        System.out.println("Pop " + deque.pop());
        System.out.println("After pop: " + deque);
        // We can check if a specific element exists in the deque
        System.out.println("Contains element 3: " +
                        deque.contains("Element 3 (Tail)"));
      // We can remove the first / last element.
        deque.removeFirst();
        deque.removeLast();
        System.out.println("Deque after removing " +
                            "first and last: " + deque);

    }
}
```

Output:

```
[Element 6 (Head), Element 4 (Head), Element 2 (Head), Element 1 (Tail), Element 3 (Tail),
 Element 5 (Tail), Element 7 (Tail)]

Standard Iterator
    Element 6 (Head)
    Element 4 (Head)
    Element 2 (Head)
    Element 1 (Tail)
    Element 3 (Tail)
    Element 5 (Tail)
    Element 7 (Tail)
Reverse Iterator
    Element 7 (Tail)
    Element 5 (Tail)
    Element 3 (Tail)
    Element 1 (Tail)
    Element 2 (Head)
    Element 4 (Head)
    Element 6 (Head)
Peek Element 6 (Head)
After peek: [Element 6 (Head), Element 4 (Head), Element 2 (Head), Element 1 (Tail),
 Element 3 (Tail), Element 5 (Tail), Element 7 (Tail)]
Pop Element 6 (Head)
After pop: [Element 4 (Head), Element 2 (Head), Element 1 (Tail), Element 3 (Tail),
 Element 5 (Tail), Element 7 (Tail)]
Contains element 3: true
Deque after removing first and last: [Element 2 (Head), Element 1 (Tail), Element 3 (Tail),
 Element 5 (Tail)]
```

Deque in Collection Hierarchy :



9. Comparator vs Comparable

Using Comparable Interface

A comparable object is capable of comparing itself with another object. The class itself must implements the **java.lang.Comparable** interface to compare its instances.
Consider a Movie class that has members like, rating, name, year. Suppose we wish to sort a list of Movies based on year of release. We can implement the Comparable interface with the Movie class, and we override the method compareTo() of Comparable interface.

```
// A Java program to demonstrate use of Comparable
import java.io.*;
import java.util.*;
// A class 'Movie' that implements Comparable
class Movie implements Comparable<Movie>
{
    private double rating;
    private String name;
    private int year;

    // Used to sort movies by year
    public int compareTo(Movie m)
    {
        return this.year - m.year;
    }
   // Constructor
    public Movie(String nm, double rt, int yr)
    {
        this.name = nm;
        this.rating = rt;
        this.year = yr;
    }
 // Getter methods for accessing private data
    public double getRating() { return rating; }
    public String getName()   {  return name; }
    public int getYear()      {  return year;  }
}
 // Driver class
class Main
{
    public static void main(String[] args)
    {
        ArrayList<Movie> list = new ArrayList<Movie>();
        list.add(new Movie("Force Awakens", 8.3, 2015));
        list.add(new Movie("Star Wars", 8.7, 1977));
        list.add(new Movie("Empire Strikes Back", 8.8, 1980));
        list.add(new Movie("Return of the Jedi", 8.4, 1983));
    Collections.sort(list);
  System.out.println("Movies after sorting : ");
        for (Movie movie: list)
        {
            System.out.println(movie.getName() + " " +
                               movie.getRating() + " " +
                               movie.getYear());
        }
    }
```

```
}
```

Output:

```
Movies after sorting :
Star Wars 8.7 1977
Empire Strikes Back 8.8 1980
Return of the Jedi 8.4 1983
Force Awakens 8.3 2015
```

Now, suppose we want sort movies by their rating and names also. When we make a collection element comparable(by having it implement Comparable), we get only one chance to implement the compareTo() method. The solution is using Comparator.

Using Comparator

Unlike Comparable, Comparator is external to the element type we are comparing. It's a separate class. We create multiple separate classes (that implement Comparator) to compare by different members.

Collections class has a second sort() method and it takes Comparator. The sort() method invokes the compare() to sort objects.

To compare movies by Rating, we need to do 3 things :

1. Create a class that implements Comparator (and thus the compare() method that does the work previously done by compareTo()).
2. Make an instance of the Comparator class.
3. Call the overloaded sort() method, giving it both the list and the instance of the class that implements Comparator.

```
//A Java program to demonstrate Comparator interface
import java.io.*;
import java.util.*;
// A class 'Movie' that implements Comparable
class Movie implements Comparable<Movie>
{
   private double rating;
   private String name;
   private int year;
  // Used to sort movies by year
   public int compareTo(Movie m)
   {
      return this.year - m.year;
   }
```

```
    // Constructor
    public Movie(String nm, double rt, int yr)
    {
        this.name = nm;
        this.rating = rt;
        this.year = yr;
    }
    // Getter methods for accessing private data
    public double getRating() { return rating; }
    public String getName()   {  return name; }
    public int getYear()      {  return year;  }
}
 // Class to compare Movies by ratings
class RatingCompare implements Comparator<Movie>
{
    public int compare(Movie m1, Movie m2)
    {
        if (m1.getRating() < m2.getRating()) return -1;
        if (m1.getRating() > m2.getRating()) return 1;
        else return 0;
    }
}

// Class to compare Movies by name
class NameCompare implements Comparator<Movie>
{
    public int compare(Movie m1, Movie m2)
    {
        return m1.getName().compareTo(m2.getName());
    }
}
 // Driver class
class Main
{
```

```
public static void main(String[] args)
{
    ArrayList<Movie> list = new ArrayList<Movie>();
    list.add(new Movie("Force Awakens", 8.3, 2015));
    list.add(new Movie("Star Wars", 8.7, 1977));
    list.add(new Movie("Empire Strikes Back", 8.8, 1980));
    list.add(new Movie("Return of the Jedi", 8.4, 1983));

    // Sort by rating : (1) Create an object of ratingCompare
    //                  (2) Call Collections.sort
    //                  (3) Print Sorted list
    System.out.println("Sorted by rating");
    RatingCompare ratingCompare = new RatingCompare();
    Collections.sort(list, ratingCompare);
    for (Movie movie: list)
        System.out.println(movie.getRating() + " " +
                           movie.getName() + " " +
                           movie.getYear());

   // Call overloaded sort method with RatingCompare
    // (Same three steps as above)
    System.out.println("\nSorted by name");
    NameCompare nameCompare = new NameCompare();
    Collections.sort(list, nameCompare);
    for (Movie movie: list)
        System.out.println(movie.getName() + " " +
                           movie.getRating() + " " +
                           movie.getYear());

    // Uses Comparable to sort by year
    System.out.println("\nSorted by year");
    Collections.sort(list);
    for (Movie movie: list)
        System.out.println(movie.getYear() + " " +
                           movie.getRating() + " " +
                              movie.getName()+" ");
    }
}
```

Output :

```
Sorted by rating
8.3 Force Awakens 2015
8.4 Return of the Jedi 1983
8.7 Star Wars 1977
8.8 Empire Strikes Back 1980

Sorted by name
Empire Strikes Back 8.8 1980
Force Awakens 8.3 2015
Return of the Jedi 8.4 1983
Star Wars 8.7 1977

Sorted by year
1977 8.7 Star Wars
1980 8.8 Empire Strikes Back
1983 8.4 Return of the Jedi
2015 8.3 Force Awakens
```

- Comparable is meant for objects with natural ordering which means the object itself must know how it is to be ordered. For example Roll Numbers of students. Whereas, Comparator interface sorting is done through a separate class.
- Logically, Comparable interface compares “this” reference with the object specified and Comparator in Java compares two different class objects provided.
- If any class implements Comparable interface in Java then collection of that object either List or Array can be sorted automatically by using Collections.sort() or Arrays.sort() method and objects will be sorted based on there natural order defined by CompareTo method.

To summarize, if sorting of objects needs to be based on natural order then use Comparable whereas if you sorting needs to be done on attributes of different objects, then use Comparator in Java.

10. Fail Fast vs Fail Safe Iterator

What is Concurrent Modification ?

When one or more thread is iterating over the collection, in between, one thread changes the structure of the collection (either adding the element to the collection or by deleting the element in the collection or by updating the value at particular position in the collection) is known as Concurrent Modification.

Difference between Fail Fast iterator and Fail Safe iterator

Fail fast Iterator

Fail fast iterator while iterating through the collection , instantly throws Concurrent Modification Exception if there is structural modification of the collection .

Fail-fast iterator can throw ConcurrentModificationException in two scenarios :

Single Threaded Environment
After the creation of the iterator , structure is modified at any time by any method other than iterator's own remove method.

Multiple Threaded Environment
If one thread is modifying the structure of the collection while other thread is iterating over it .

Interviewer : How Fail Fast Iterator come to know that the internal structure is modified ?
All Collection types maintain an internal array of objects (Object[]) to store the elements. *Fail-Fast* iterators directly fetch the elements from this array. They always consider that this internal array is not modified while iterating over its elements. To know whether the collection is modified or not, they use an internal flag called *modCount* which is updated each time a collection is modified. Every time when an Iterator calls the *next()* method, it checks the *modCount*. If it finds the *modCount* has been updated after this Iterator has been created, it throws *ConcurrentModificationException*.

The iterators returned by the *ArrayList*, *Vector*, *HashMap* etc are all *Fail-Fast* in nature.

Fail Safe Iterator :

Fail Safe Iterator makes copy of the internal data structure (object array) and iterates over the copied data structure.Any structural modification done to the iterator affects the copied data structure. So , original data structure remains structurally unchanged .Hence , no ConcurrentModificationException throws by the fail safe iterator.

Two issues associated with Fail Safe Iterator are :

1. Overhead of maintaining the copied data structure i.e memory.

2. Fail safe iterator does not guarantee that the data being read is the data currently in the original data structure.

But, these iterators have some drawbacks. One of them is that it is not always guaranteed that you will get up-to-date data while iterating. Because any modifications to collection after the iterator has been created is not updated in the iterator. One more disadvantage of these iterators is that there will be additional overhead of creating the copy of the collection in terms of both time and memory.

Iterator returned by *ConcurrentHashMap* is a fail-safe iterator.

Example of Fail Fast Iterator and Fail Safe Iterator

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class FailFastExample
{
  public static void main(String[] args)
   {
      Map<String,String> premiumPhone = new HashMap<String,String>();
      premiumPhone.put("Apple", "iPhone");
      premiumPhone.put("HTC", "HTC one");
      premiumPhone.put("Samsung","S5");

      Iterator iterator = premiumPhone.keySet().iterator();
      while (iterator.hasNext())
      {
         System.out.println(premiumPhone.get(iterator.next()));
         premiumPhone.put("Sony", "Xperia Z");
      }
   }
 }
```

Output :

```
iPhone
Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
        at java.util.HashMap$KeyIterator.next(Unknown Source)
        at FailFastExample.main(FailFastExample.java:20)
```

Fail Safe Iterator Example :

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.Iterator;
public class FailSafeExample
{
  public static void main(String[] args)
    {
        ConcurrentHashMap<String,String> premiumPhone =
                              new ConcurrentHashMap<String,String>();
        premiumPhone.put("Apple", "iPhone");
        premiumPhone.put("HTC", "HTC one");
        premiumPhone.put("Samsung","S5");
        Iterator iterator = premiumPhone.keySet().iterator();
         while (iterator.hasNext())
        {
            System.out.println(premiumPhone.get(iterator.next()));
            premiumPhone.put("Sony", "Xperia Z");
```

```
      }
    }
  }
```

Output :

```
S5
HTC one
iPhone
```