**Collection Framework Introduction:**

The Java collections framework provides a set of inter faces and classes to implement various data structures and algorithms.

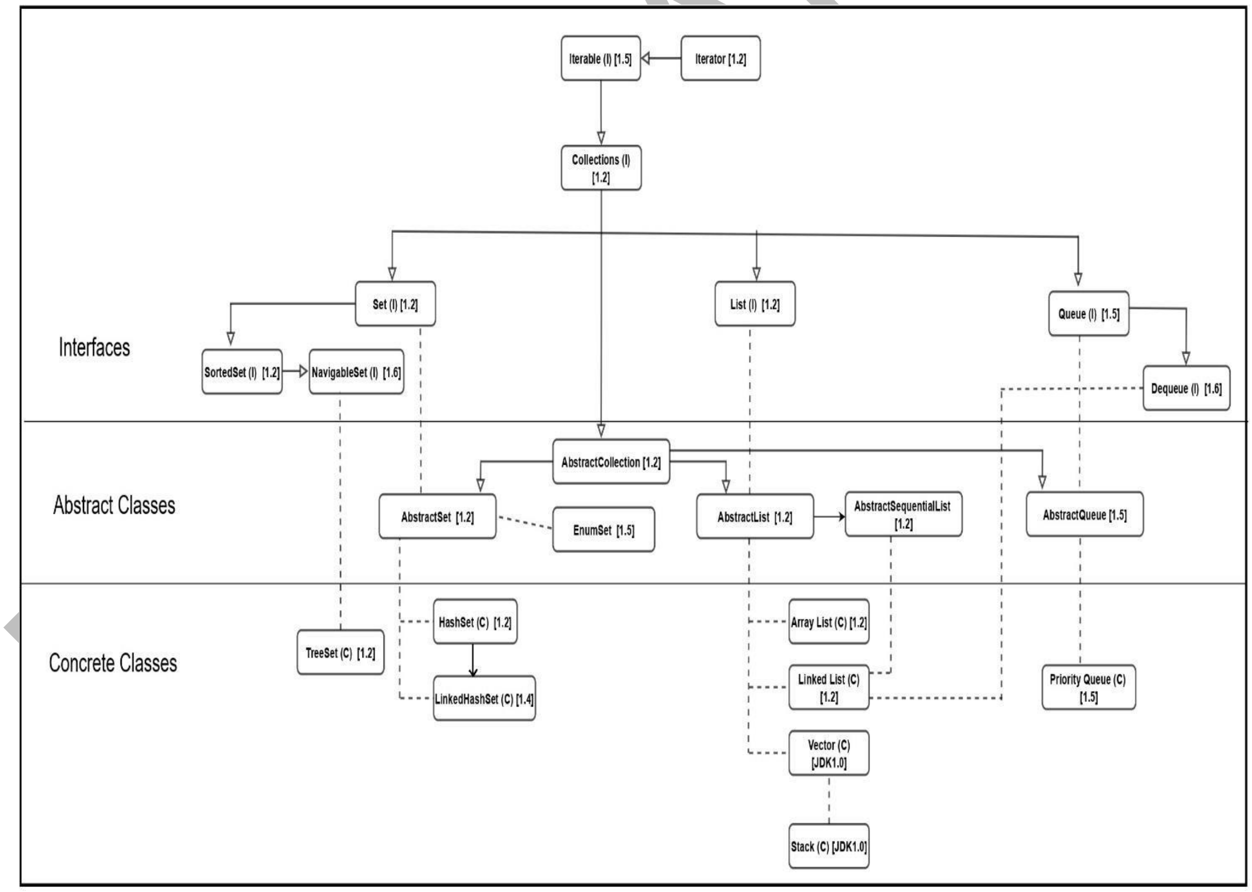
It provides an organized ready-made architecture for interfaces and classes which is used for storing and manipulating a group of objects. A collection is a group of objects or it is a single entity that represents multiple objects.

**Legacy classes:**

Collection is a predefined interface which came from java 1.2 version. Before Collection interface, in order to work with single object, we had Vector class and for working with group of objects we had Hashtable class which is the sub class for Dictionary class (abstract class). These classes are known as Legacy classes.

All of these classes had no common interface and working independently Therefore, it was very difficult for developer to work independently as well as to remember all the different methods, syntax, and constructor present in every collection class so, from java 1.2 version they decided to re-structure everything inside a single umbrella and introduced two interfaces i.e., Collection and Map to work with individual and group of objects respectively.

**Collection Hierarchy:**



Collection interface provides the following sub interfaces:

1) List (Accept duplicate elements)
2) Set (Not accepting duplicate elements)
3) Queue (Storing and Fetching the elements based on some order i.e., FIFO)

**Methods of Collection interface:**

a. public boolean add(Object element) :- Used to add an element in thecollection.
b. public boolean addAll(Collection c) :- It is used to insert the specified collection elements in the existing collection.
c. public boolean remove(Object element) :- It is used to delete an element from the collection.
d. public boolean removeAll(Collection c) :- It is used to delete all the elements from the existing collection.
e. public boolean retainAll(Collection c) :- It is used to retain all the elements from existing element.
f. public int size() :- It is used to find out the size of the Collection.
g. public void clear() :- It is used to clear all the elements at once from the Collection.
h. public boolean contains() :- It will return true if the collection contains specified element.
i. public boolean containsAll(Collection c) : It will return true if the collection contains all the elements in the given collection.
j. public Iterator iterator() :- It is used to iterate or retrieve the elements from the collection.
k. public boolean removeIf(Predicate p) :- It will remove all the elements, if it satisfy the given predicate.
l. public Spliteratorspliterator() :- It is used to create spliterator over the collection.
m. public Stream stream() :- It is used to return a sequential Stream with this collection as its source.
n. public Stream parallelStream() :- It is used to return a parallel Stream with this collection as its source.
o. public T toArray() :- This method is used to return an array containing all of the elements in this collection.

Note: All the above said methods will be applicable to all the sub interfaces of Collection like List, Set and Queue.

**List interface:**

It is the sub interface of Collection interface.

It can accept duplicate elements.

We can perform sorting Operation on List interface manually.

It stores the elements on the basis of index just like an array.

**Methods of List interface:**

Here is a list of some commonly used methods of the List interface:

a) public boolean add(E element): Adds the specified element to the end of the list.
b) public void add(int index, E element): Inserts the specified element at the specified position in the list.

c) public boolean addAll(Collection<? extends E> c): Adds all elements of the specified collection to the end of the list.
d) public boolean addAll(int index, Collection<? extends E> c): Inserts all elements of the specified collection at the specified position in the list.
e) public void clear(): Removes all elements from the list.
f) public boolean contains(Object o):Returns true if the list contains the specified element.
g) public Object get(int index): Returns the element at the specified position in the list.
h) public int indexOf(Object o): Returns the index of the first occurrence of the specified element in the list.
i) public boolean isEmpty(): Returns true if the list contains no elements.
j) public Iterator iterator(): Returns an iterator over the elements in the list.
k) public boolean remove(Object o): Removes the first occurrence of the specified element from the list.
l) public E remove(int index): Removes the element at the specified position in the list.
m) public boolean removeAll(Collection<?> c): Removes all elements in the specified collection from the list.
n) public boolean retainAll(Collection<?> c): Retains only the elements in the list that are contained in the specified collection.
o) public int size(): Returns the number of elements in the list.
p) public List subList(int fromIndex, int toIndex): Returns a view of the portion of the list between the specified fromIndex (inclusive) and toIndex (exclusive).
q) public Object toArray(): Returns an array containing all of the elements in the list.
r) public T[] toArray(T[] a): Returns an array containing all of the elements in the list; the runtime type of the returned array is that of the specified array.

**Behaviour of List interface Specific classes:**

a. It stores heterogeneous types of data.
b. It stores the elements based on the index position.
c. It stores everything in the form of Object.
d. We can implement generic(<>) to remove the compilation warning and accepting type safe objects.
e. By using generic, we can also take heterogeneous types of elements by using <Object>
f. IT IS DYNAMICALLY GROWABLE ARRAY.
g. We can accept null values.
h. We can perform sorting operation.

**How many ways we can fetch Collection Object from Collections Framework:**

There are 7 ways to fetch these Collection object from Collections Framework

1) By Using Enumeration interface (JDK 1.0)
2) By using Ordinary for loop
3) By using for-Each loop
4) By using Iterator interface

*5) By using ListIterator interface

*6) By using forEach(Consumer cons) Method

*7) By using Method Reference (::)

Note: Among all these 7, Enumeration, Iterator and ListIterator these are cursors

**Enumeration interface:**

It is a predefined interface available in java.util package from JDK 1.0 onwards.

We can use Enumeration interface to fetch or retrieve the Objects one by one from the Collection because it is a cursor.

We can create Enumeration object by using elements() method of the respective Collection class.

public Enumeration elements();

Enumeration interface contains two methods:

a) public boolean hasMoreElements() :- It will return true if the Collection is having more elements.
b) public Object nextElement() :- It will return collection object so return type is Object.

**Iterator interface:**

It is a predefined interface available in java.util package available from 1.2 version.

It is used to fetch/retrieve the elements from the Collection in forward direction only.

public Iterator iterator();

Example:

Iterator itr = v.iterator();

Now, Iterator interface has provided two methods

a) public boolean hasNext():It will verify the element is available in the next position or not, if available it will return true otherwise it will return false.
b) public Object next() :- It will return the collection object.

**ListIterator interface:**

It is a predefined interface available in java.util package and it is the sub interface of Iterator.

It is used to retrieve the Collection object in both the direction i.e. in forward direction as well as in backward direction.

public ListIterator listIterator();

Example:

ListIterator lit = v.listIterator();

Now, ListIterator interface has provided the following methods:

a) public boolean hasNext(): It will verify the element is available in the next position or not, if available it will return true otherwise it will return false.
b) public Object next(): It will return the next position collection object.

c) public boolean hasPrevious(): It will verify the element is available in the previous position or not, if available it will return true otherwise it will return false.
d) public Object previous (): It will return the previous position collection object.

Note: Apart from these 4 methods we have add(), set() and remove() methods in ListIterartor interface.

**By using forEach() method:**

From java 1.8 onwards every collection class provides the method forEach() method, this method takes Consumer functional interface as a parameter.

The following program explains how forEach(Consumer cons) method works internally.

```
package com.ravi.intro;
import java.util.Vector;
import java.util.function.Consumer;

public class ForEachWorking {

        public static void main(String[] args)
        {
                Vector<String> fruits = new Vector<>();
                fruits.add("Orange");
                fruits.add("Apple");
                fruits.add("Mango");
                fruits.add("Banana");
                fruits.add("Gauva");

                //Anonymous inner class
                Consumer<String> cons = new Consumer<String>()
          {
                        @Override
                        public void accept(String t)
                        {
                                System.out.println(t);
                        }
                };

                fruits.forEach(cons);
        }

}
```

**The following program explains how many ways we can fetch Collection Object(All 7 ways).**

```
package com.ravi.intro;
import java.util.Collections;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.Vector;

public class FetchingCollectionData
{
        public static void main(String[] args)
        {
                Vector<String> fruits = new Vector<>();
                fruits.add("Orange");
                fruits.add("Apple");
                fruits.add("Mango");
                fruits.add("Banana");
                fruits.add("Gauva");
        Collections.sort(fruits);
                System.out.println("FETCHING THE DATA FROM ENUMERATION INTERFACE");

    Enumeration<String> elements = fruits.elements();
    while(elements.hasMoreElements())
    {
         System.out.println(elements.nextElement());
    }
    System.out.println("FETCHING THE DATA FROM ORDINARY FOR LOOP");
       for(int i=0; i<fruits.size(); i++)
{
        System.out.println(fruits.get(i));
}
    System.out.println("FETCHING THE DATA FROM FOR EACH LOOP");
       for(String fruit : fruits)
 {
         System.out.println(fruit);
     }
     System.out.println("FETCHING THE DATA FROM ITERATOR INTERFACE");
       Iterator<String> itr = fruits.iterator();
    while(itr.hasNext())
     {
 System.out.println(itr.next());
     }
     System.out.println("FETCHING THE DATA FROM LISTITERATOR INTERFACE");
    ListIterator<String> litr = fruits.listIterator();
    System.out.println("IN FORWARD DIRECTION");
    while(litr.hasNext())
     {
          System.out.println(litr.next());
     }

     System.out.println("IN BACKWARD DIRECTION");
```

```
        while(litr.hasPrevious())
        {
                System.out.println(litr.previous());
        }
        System.out.println("FETCHING THE DATA FROM FOREACH METHOD");
        fruits.forEach(fruit->System.out.println(fruit));
        System.out.println("FETCHING THE DATA USING METHOD REFERENCE");
        fruits.forEach(System.out::println);
        }
}
```

**List Implementer classes:**

**ArrayList:**

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

- It is a predefined class available in java.util package under List interface.
- It accepts duplicate elements and null values.
- It is dynamically growable array.
- It stores the elements on index basis so it is simillar to dynamic array.
- Initial capacity of ArrayList is 10. The new capacity of Arraylist can be calculated by using the formula new capacity = (current capacity * 3/2) + 1
- All the methods declared inside an ArrayList is not synchronized so multiple thread can access the method of ArrayList.
- It is highly suitable for fetching or retriving operation when duplicates are allowed and Thread-safety is not required.
- Implements the following interfaces:

➢ List: Allows for the manipulation of elements based on their index.
➢ Serializable: Allows ArrayList instances to be converted into a byte stream for serialization.
➢ Cloneable: Supports the creation of a shallow copy of the ArrayList.
➢ RandomAccess: Indicates that the ArrayList supports fast, random access to its elements.

**Constructor of ArrayList :**

In ArrayList we have 3 types of Constructor:

1) ArrayList al1 = new ArrayList();

Will create ArrayList object with default capacity 10.

2) ArrayList al2 = new ArrayList(int initialCapacity);

Will create an ArrayList object with user specified Capacity

3)ArrayList al3 = new ArrayList(Collection c);

We can copy any Collection interface implemented class data to the current object reference (Coping one Collection data to another).

**Array List Methods :**

a) public boolean add(E eleme) : Adds the specified element to the end of this list.
b) public void add(int index, E element) : Inserts the specified element at the specified position in this list.
c) public boolean addAll(Collection<? Extends E> c) : Adds all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
d) public boolean addAll(int index, Collection<? extends E> c) : Inserts all of the elements in the specified collection into this list, starting at the specified position.
e) public void clear() : Removes all of the elements from this list.
f) public boolean contains(Object o) : Returns true if this list contains the specified element.
g) public void ensureCapacity(int minCapacity) : Increases the capacity of this Array List instance.
h) public E get(int index) : Returns the element at the specified position in current list.
i) public int indexOf)Object o) : Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
j) public boolean isEmpty() : Returns true if this list contains no elements.
k) public boolean remove(int index) : Removes the element at the specified position in this list.
l) public boolean remove(Object 0) : Removes the first occurrence of the specified element from this list, if it is present.
m) public boolean removeAll(Collection<?> c) : Removes from this list all of its elements that are contained in the specified collection.
n) public E set(int index, E element) : Replaces the element at the specified position in this list with the specified element.
o) public int size() : Returns the number of elements in this list.
p) public Object[] toArray() : Returns an array containing all of the elements in this list in proper sequence (from first to last element).

**Creating ArrayList by using different Constructors :**

```
import java.util.ArrayList;

public class ArrayListExample {
  public static void main(String[] args) {
    // Using default constructor
    ArrayList<String> list1 = new ArrayList<>();
    list1.add("Ram");
    list1.add("Anil");
    list1.add("Sita");
System.out.println(list1);
    // Using constructor with initial capacity
    ArrayList<String> list2 = new ArrayList<>(10);
    list2.add("Apple");
    list2.add("Banana");
    list2.add("Orange");
    System.out.println(list2);

    // Using constructor with another collection
    ArrayList<String> originalList = new ArrayList<>();
```

```
        originalList.add("Java");
        originalList.add("Python");
        originalList.add("C++");

        ArrayList<String> list3 = new ArrayList<>(originalList);
        System.out.println(list3);
    }
}
```

**Example programs on ArrayList :**

**Operations** : finding size, adding element, removing element based on index position, removing element based on specific Object and sorting the elements.

```
public class ArrayListDemo
{
        public static void main(String... a)
        {
                ArrayList<String> arl = new ArrayList<>();//Generic type
                arl.add("Apple");
                arl.add("Orange");
                arl.add("Grapes");
                arl.add("Mango");
                arl.add("Guava");
                arl.add("Mango");

          System.out.println("Size :"+arl.size());
                System.out.println("Contents :"+arl); //toString()  [Apple,....]

                arl.remove(2); //based on the index position
                arl.remove("Guava"); //based on the Object

                System.out.println("Contents After Removing :"+arl);
                System.out.println("Size of the ArrayList:"+arl.size());

                Collections.sort(arl);

          arl.forEach( x -> System.out.println(x));
        }
}
```

**Operations** : storing custom object in an ArrayList

<u>Employee.java</u>

```
public class Employee
{
  private Integer employeeId;
  private String employeeName;
private Double employeeSalary;
public Employee() {
        super();
        // TODO Auto-generated constructor stub
  }
public Employee(Integer employeeId, String employeeName, Double employeeSalary) {
        super();
        this.employeeId = employeeId;
        this.employeeName = employeeName;
        this.employeeSalary = employeeSalary;
}
public Integer getEmployeeId() {
        return employeeId;
}
public void setEmployeeId(Integer employeeId) {
        this.employeeId = employeeId;
}
public String getEmployeeName() {
        return employeeName;
}
public void setEmployeeName(String employeeName) {
        this.employeeName = employeeName;
}

public Double getEmployeeSalary() {
        return employeeSalary;
}
public void setEmployeeSalary(Double employeeSalary) {
        this.employeeSalary = employeeSalary;
}
@Override
public String toString() {
        return "Employee [employeeId=" + employeeId + ", employeeName=" + employeeName + ",
employeeSalary="+ employeeSalary + "]";
}

}
```

<u>ArrayListDemo1.java</u>

```
import java.util.ArrayList;

import java.util.Iterator;


public class ArrayListDemo1

{

        public static void main(String[] args)

        {

                ArrayList<Employee> arl = new ArrayList<>();

                arl.add(new Employee(101, "Ravi", 12000.89));

                arl.add(new Employee(102, "Raj",  12000.89));
```

```
                    arl.add(new Employee(103, "Aryan", 12000.89));

                    arl.add(new Employee(104, "Virat", 12000.89));


                    arl.forEach(emp->System.out.println(emp));

            }

}
```

**Operation**: fetching the elements in forward and backward direction using ListIterator.

```
      import java.util.*;
      public class ArrayListDemo3
      {
      public static void main(String args[])
            {
               //Arrays class is having static method asList()
                        List<String> list=  Arrays.asList("Ravi","Rahul","Sweta","Ananya","Bina");

                        Collections.sort(list);
                        Collections.reverse(list);

                        ListIterator<String> itr=list.listIterator();

                        System.out.println("traversing elements in forward direction...");
                        while(itr.hasNext())
                         {
                                    System.out.println(itr.next());
                         }

                        System.out.println("traversing elements in backward direction...");
                        while(itr.hasPrevious())
                         {
                        System.out.println(itr.previous());
                         }
            }
}
```

**Operation**: resizing the capacity of ArrayList Object, adding element based on specific index, using forEach and Lambda to Modify elements.

```
import java.util.ArrayList;

public class ArrayListDemo {
   public static void main(String[] args) {
       // Resizing ArrayList using ensureCapacity
       ArrayList<String> city = new ArrayList<>(); // Default capacity is 10
       city.ensureCapacity(3); // Resized the ArrayList to store 3 elements.

       city.add("Hyderabad");
```

```
        city.add("Mumbai");
        city.add("Delhi");

        city.add("Kolkata");
        System.out.println("ArrayList: " + city);

        // ArrayList with null values and inserting elements based on index
        ArrayList<Object> al = new ArrayList<>(); // Generic type
        al.add(12);
        al.add("Ravi");
        al.add(12);
        al.add(3, "Hyderabad"); // add(int index, Object o) method of List interface
        al.add(1, "Naresh");
        al.add(null);
        al.add(11);
        System.out.println(al);

        // Using forEach and Lambda Expression to Modify Elements
        ArrayList<Integer> numbers = new ArrayList<>();
        for (int i = 1; i <= 5; i++) {
            numbers.add(i);
        }

        // Using forEach and Lambda Expression to double each element
        System.out.print("Modified ArrayList: ");
        numbers.forEach(x -> System.out.print(x * 2 + " "));
    }
}
```

**Operation**: Serialization and De-serialization on ArrayList Object.

Product.java

```
package com.ravi.serialization_on_Arraylist;

import java.io.Serializable;

public class Product implements Serializable
{
  private Integer productId;
  private String productName;

        public Product(Integer productId, String productName)
        {
                super();
                this.productId = productId;
                this.productName = productName;
        }

        public Integer getProductId() {
                return productId;
        }

        public void setProductId(Integer productId) {
                this.productId = productId;
        }

        public String getProductName() {
                return productName;
        }

        public void setProductName(String productName) {
                this.productName = productName;
        }

        @Override
        public String toString() {
                return "Product [productId=" + productId + ", productName=" + productName + "]";
        }
}
```

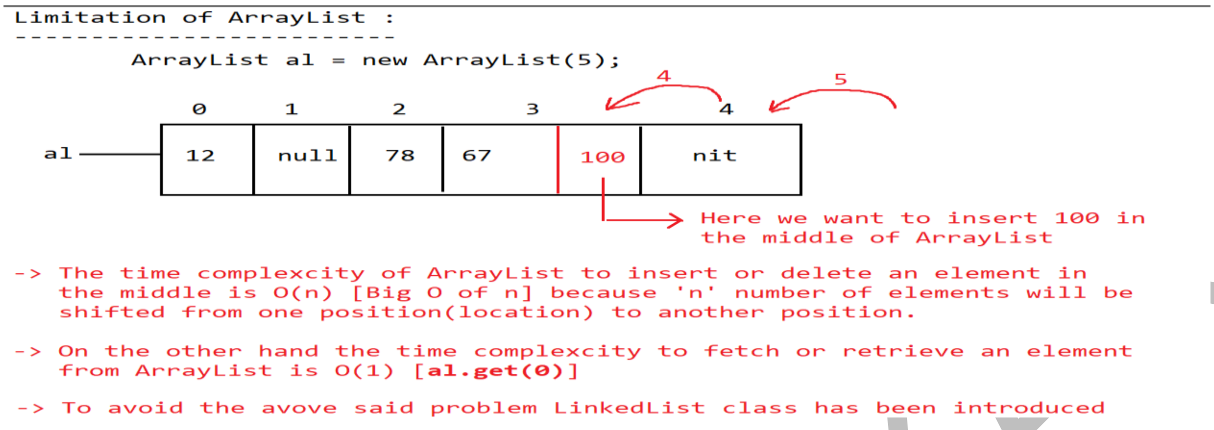
ArrayListSerializationAndDeSerialization.java

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

class ArrayListSerializationAndDeSerialization {
   public static void main(String[] args) {
      // Serialization and De-serialization on ArrayList<String>
      try {
         ArrayList<String> al = new ArrayList<>();
         al.add("Nagpur");
         al.add("Vijaywada");
         al.add("Hyderabad");
         al.add("Jamshedpur");

         // Serialization
         FileOutputStream fos = new FileOutputStream("D:\\new\\City.txt");
         ObjectOutputStream oos = new ObjectOutputStream(fos);
         oos.writeObject(al);

         // De-serialization
         FileInputStream fis = new FileInputStream("D:\\new\\City.txt");
         ObjectInputStream ois = new ObjectInputStream(fis);

         ArrayList<String> list = (ArrayList<String>) ois.readObject();

         System.out.println("Serialized and De-serialized ArrayList<String>: " + list);

         // Close streams
         fos.close();
         oos.close();
         fis.close();
         ois.close();
```

```
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
        // Serialization and De-serialization on ArrayList<Product>
        try {
            ArrayList<Product> productList = new ArrayList<>();
            productList.add(new Product(111, "Laptop"));
            productList.add(new Product(222, "I Phone"));
            productList.add(new Product(333, "Mobile"));

            // Serialization
            FileOutputStream fos = new FileOutputStream("D:\\new\\Product.txt");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(productList);
            System.out.println("Product Data stored successfully");

            // De-serialization
            FileInputStream fin = new FileInputStream("D:\\new\\Product.txt");
            ObjectInputStream ois = new ObjectInputStream(fin);
            Object prod = ois.readObject();

            System.out.println("Serialized and De-serialized ArrayList<Product>: " + prod);

            // Close streams
            fos.close();
            oos.close();
            fin.close();
            ois.close();

        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

**Limitation of ArrayList:**

The time complexity of ArrayList to insert and delete an element from the middle would be O(n) because 'n' number of elements will be re-located so it is not a good choice to perform insertion and deletion operation in the middle of the List. To avoid this we introduced LinkedList as an alternative.



```
Limitation of ArrayList :
-------------------------
        ArrayList al = new ArrayList(5);
```

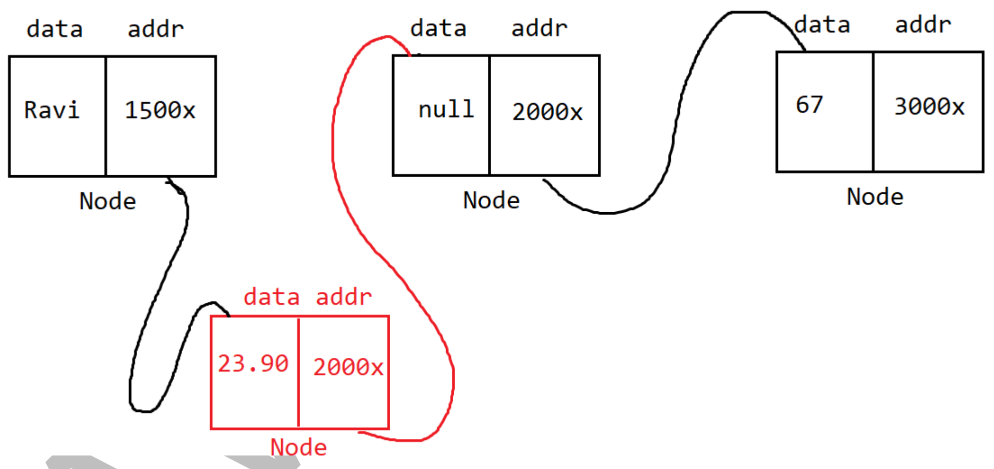
-> The time complexcity of ArrayList to insert or delete an element in the middle is O(n) [Big O of n] because 'n' number of elements will be shifted from one position(location) to another position.

-> On the other hand the time complexcity to fetch or retrieve an element from ArrayList is O(1) [**al.get(0)**]

-> To avoid the avove said problem LinkedList class has been introduced

**LinkedList:**

LinkedList :-
------------



- public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable
- It is a predefined class available in java.util package under List interface.
- It is ordered by index position like ArrayList except the elements (nodes) are doubly linked to one another. This linkage provide us new method for adding and removing the elements from the middle of LinkedList.
- The important thing is, LikedList may iterate more slowely than ArrayList but LinkedList is a good choice when we want to insert or delete the elements frequently in the list.
- From jdk 1.5 onwards LinkedList class has been enhanced to support basic queue operation by implementing Deque<E> interface.
- ArrayList is using Array data structure but LinkedList class is using LinkedList data structure.

**Constructors of Linked List:**

5) public LinkedList();
It will create a LinkedList object with 0 capacity.

6) public LinkedList(Collection<? extends E> c);
Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

**Methods of LinkedList:**

a) public E getFirst() : Returns the first element in current list.
b) public E getLast() : Returns the last element in current list.
c) public E removeFirst() : Removes and returns the first element from current list.
d) public E removeLast() : Removes and returns the last element from current list.
e) public void addFirst(E e) : Inserts the specified element at the beginning of current list.
f) public void addLast(E e) : Adds the specified element to the end of current list.
g) public boolean contains(Object o) : Returns true if current list contains the specified element.
h) public int size() : Returns the number of elements in current list.
i) public boolean add(E e) : Adds the specified element to the end of current list.
j) public boolean remove(Object o) : Removes the first occurrence of the specified element from current list, if it is present. If current list does not contain the element, it is unchanged.
k) public boolean addAll(Collection<? extends E> c) : Adds all of the elements in the specified collection to the end of the current list, in the order that they are returned by the specified collection's iterator.
l) public boolean addAll(int index, Collection<? extends E> c) : Inserts all of the elements in the specified collection into the current list, starting at the specified position.
m) public void clear() : Removes all of the elements from the current list. The list will be empty after this call returns.
n) public E get(int index) : Returns the element at the specified position in the current list.
o) public E set(int index, E element) : Replaces the element at the specified position in the current list with the specified element.
p) public void add(int index, E element) : Inserts the specified element at the specified position in the current list. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).
q) public E remove(int index) : Removes the element at the specified position in the current list.
r) public int indexOf(Object o) : Returns the index of the first occurrence of the specified element in this list, or -1 if the current list does not contain the element.
s) public E peek() : Retrieves, but does not remove, the head (first element) of the current list.
t) public E element() : Retrieves, but does not remove, the head (first element) of the current list.
u) public E poll() : Retrieves and removes the head (first element) of the current list.
v) public E remove() : Retrieves and removes the head (first element) of the current list.

**[Note:-It stores the elements in non-contiguous memory location. The time complexity for insertion and deletion is O(1). The time complexity for searching O(n).]**

**Example programs on LinkedList:**

**Operations**: Storing elements, fetching elements by using java8(forEach()), replacing element, adding elements based on specific position.

```
import java.util.LinkedList;

public class LinkedListOperations {
  public static void main(String[] args) {
    // Storing elements in LinkedList
    LinkedList<String> linkedList = new LinkedList<>();
    linkedList.add("Apple");
    linkedList.add("Orange");
    linkedList.add("Litchi");
    linkedList.add("Grapes");

    // Fetching elements using forEach() in Java 8
    System.out.println("Elements in the LinkedList:");
    linkedList.forEach(System.out::println);

    // Replacing the second element
    int indexToReplace = 1;
    String newElement = "NewElement";
    linkedList.set(indexToReplace, newElement);// //set() will replace the existing value
    System.out.println("After replacing element at index " + indexToReplace + ":");
    linkedList.forEach(System.out::println);

    // Adding an element at the third position
    int indexToAdd = 2;
    String elementToAdd = "InsertedElement";
    linkedList.add(indexToAdd, elementToAdd);
    System.out.println("After adding element at index " + indexToAdd + ":");
    linkedList.forEach(System.out::println);
  }
}
```

**Operation**: adding elements, adding element at first, adding element at last, fetching first element, fetching last element, remove first element, remove last element.

```
import java.util.LinkedList;

public class LinkedListOperations {
  public static void main(String[] args) {
    // Creating a LinkedList
    LinkedList<String> linkedList = new LinkedList<>();

    // Adding elements to the LinkedList
    linkedList.add("Ravi");
    linkedList.add("Amar");
    linkedList.add("Amit");
    linkedList.add("Ram");

    // Displaying the original LinkedList
    System.out.println("Original LinkedList:");
    linkedList.forEach(System.out::println);
    System.out.println();
```

```
        // Adding an element at the first position
        String elementToAddFirst = "Prashant";
        linkedList.addFirst(elementToAddFirst);

        // Adding an element at the last position
        String elementToAddLast = "Ballaya";
        linkedList.addLast(elementToAddLast);

        // Displaying the LinkedList after adding elements at first and last positions
        System.out.println("After adding elements at first and last positions:");
        linkedList.forEach(System.out::println);
        System.out.println();

        // Fetching the first and last elements
        String firstElement = linkedList.getFirst();
        String lastElement = linkedList.getLast();

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

        // Removing the first and last elements
        linkedList.removeFirst();
        linkedList.removeLast();
        // Displaying the LinkedList after removing first and last elements
        System.out.println("After removing first and last elements:");
        linkedList.forEach(System.out::println);
    }
}
```

**Operation**: Use the listIterator methods[add(), set() and remove()]

```
            import java.util.*;
            public class LinkedListDemo3
            {
                        public static void main(String[] args)
                        {
                                    LinkedList<String> city = new LinkedList<>();
                                     city.add("Kolkata");
                                     city.add("Bangalore");
                                     city.add("Hyderabad");
                                     city.add("Pune");
                                     System.out.println(city);
                                    ListIterator<String> lt = city.listIterator();
                  while(lt.hasNext())
                                     {
                                              String x = lt.next();
                                              if(x.equals("Kolkata"))
                                              {
                                               lt.remove();
                                              }
                                              else if(x.equals("Hyderabad"))
                                              {
                                              lt.add("Ameerpet");
                                              }
                                              else if(x.equals("Pune"))
                                              {
                                              lt.set("Mumbai");
                                              }
                                    }
                                    city.forEach(System.out::println); //Method Reference
                        }
            }
```

**Operation**: Using Iterator retrieve the elements and forEachRemaining()

```
    import java.util.LinkedList;
    import java.util.Iterator;
    import java.util.List;

    class Dog
    {
        public String name;
        Dog(String n)
        {
            name = n;
        }
    }
    public class LinkedListDemo5
    {
        public static void main(String[] args)
        {
            List<Dog> d = new LinkedList<>();
            Dog dog = new Dog("Tiger");
            d.add(dog);
            d.add(new Dog("Tommy"));
            d.add(new Dog("Rocky"));

            Iterator<Dog> i3 = d.iterator();
            i3.forEachRemaining(x -> System.out.println(x.name));
            System.out.println("size " + d.size());
            System.out.println("Get 1st Position Object " + d.get(1).name);
        }
    }
```

**Operation**: Create a linked List and treat is as a Deque.

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

    public class LinkedListDemo6{
       public static void main(String[] args) {
          // Create a LinkedList and treat it as a Deque
```

```
            Deque<String> deque = new LinkedList<>();

            // Adding elements to the front of the deque
            deque.addFirst("Ravi");
            deque.addFirst("Raj");

            // Adding elements to the back of the deque
            deque.addLast("Pallavi");
            deque.addLast("Sweta");


            System.out.println("Deque: " + deque);
            String first = deque.removeFirst();
            String last = deque.removeLast();
            System.out.println("Removed first element: " + first);
            System.out.println("Removed last element: " + last);
            System.out.println("Updated Deque: " + deque);
        }
    }
```

**Operation:**Insertion, deletion, displaying and exit using LinkedList with switch case.

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

public class LinkedListDemo4
{
  public static void main(String[] args)
        {
     LinkedList<Integer>linkedList = new LinkedList<>();
     Scanner scanner = new Scanner(System.in);

     while (true)
               {
System.out.println("Linked List: " + linkedList);
System.out.println("1. Insert Element");
System.out.println("2. Delete Element");
                    System.out.println("3. Display Element");
System.out.println("4. Exit");
System.out.print("Enter your choice: ");

       int choice = scanner.nextInt();
       switch (choice)
                    {
         case 1:
System.out.print("Enter the element to insert: ");
           int elementToAdd = scanner.nextInt();
linkedList.add(elementToAdd);
           break;
         case 2:
           if (linkedList.isEmpty())
                                {
```

```
System.out.println("Linked list is empty. Nothing to delete.");
                }
                                                else
                                                {
System.out.print("Enter the element to delete: ");
                    int elementToDelete = scanner.nextInt();
boolean removed = linkedList.remove(Integer.valueOf(elementToDelete));
                    if (removed)
                                                            {
System.out.println("Element " + elementToDelete + " deleted from the linked list.");
                    }
                                                            else
                                                            {
System.out.println("Element not found in the linked list.");
                    }
                }
                break;
                                        case 3:
                                                System.out.println("Elements in the linked list.");
linkedList.forEach(System.out::println);
                                            break;
            case 4:
System.out.println("Exiting the program.");
scanner.close();
System.exit(0);
            default:
System.out.println("Invalid choice. Please try again.");
          }
        }
    }
}
```

Difference between ArrayList and LinkedList.

| **Array List** | **Linked List** |
|---|---|
| Dynamic Array is used to implement ArrayList internally in JAVA. | Doubly Linked List is used to implement LinkedList internally in JAVA. |
| Array List acts only as a list because it implements a list interface. | LinkedList acts as a list as well as a queue because it implements both interfaces. |
| Preferred for storing and accessing the data. | Preferred for manipulation of data. |
| Array List provides faster random access time due to direct array indexing. | LinkedList is slower for random access but efficient for insertions and deletion in the middle. |
| Array List is less efficient for the insertions and deletions in the middle because elements need to be shifted. | LinkedList excels in insertions and deletions at any position due to its structure. |
| Storage of elements is performed at a contiguous location. | Storage of elements is performed at a non-contiguous location. |
| Being dynamic in nature, Array List is resizable. | LinkedList is implemented by pointing to elements thereby no sizing issues are faced. |
| The default capacity upon initializing is set as 10 with resizing done upon extension of the list. | The is no default capacity as insertion is performed by interlinking of the nodes and no contiguous allocation of space. |
| Insertion and deletion operations are slow in ArrayList as rearrangement is required. | LinkedList is much faster in terms of insertion and deletion operations. |
| Array List is faster when iterating sequentially through elements. | While iterating through elements LinkedList is slower due to the need to follow node references. |

**Vector:**

public class Vector<E> extends AbstractList<E> implements List<E>, Serializable, Clonable, RandomAccess

- Vector is a predefined class available in java.util package under List interface.
- Vector is always from java means it is available from jdk 1.0 version.
- Vector and Hashtable, these two classes are available from jdk 1.0, remaining Collection classes were added from 1.2 version. That is the reason Vector and Hashtable are called legacy(old) classes.
- The main difference between Vector and ArrayList is, ArrayList methods are not synchronized so multiple threads can access the method of ArrayList where as on the other hand most the methods are synchronized in Vector so performance wise Vector is slow.
- We should go with ArrayList when ThreadSafety is not required on the other hand we should go with Vector when we need ThreadSafety for retrieval operation.
- It also stores the elements on index basis. It is dynamically growable with initial capacity 10. The next capacity will be 20 i.e double of the first capacity.

new capacity = current capacity * 2;

- Just like ArrayList it also implements List, Serializable, Clonable, RandomAccess interfaces.

**Constructors in Vector:**

We have 4 types of Constructor in Vector
1) Vector v1 = new Vector();
It will create the vector object with default capacity is 10.
2) Vector v2 = new Vector(int initialCapacity);
Will create the vector object with user specified capacity.
3) Vector v3 = new Vector(int initialCapacity, int incrementalCapacity);
Eg :- Vector v = new Vector(1000,5);
Initially It will create the Vector Object with initial capacity 1000 and then when the capacity will be full then increment by 5 so the next capacity would be 1005, 1010 and so on.
4) Vector v4 = new Vector(Collection c);
Interconversion between the Collection.

**Methods Of Vector:**

a) public void copyInto(Object[] anArray): Copies the components of this vector into the specified array.
b) public void trimToSize(): Trims the capacity of this vector to be the vector's current size.
c) public void ensureCapacity(int minCapacity): Increases the capacity of this vector, if necessary.
d) public void setSize(int newSize): Sets the size of this vector.
e) public int capacity(): Returns the current capacity of this vector.
f) public int size(): Returns the number of components in this vector.
g) public boolean isEmpty(): Tests if this vector has no components.
h) public E elementAt(int index): Returns the component at the specified index.
i) public E firstElement(): Returns the first component (the item at index 0) of this vector.
j) public E lastElement(): Returns the last component of the vector.
k) public void setElementAt(E obj,int index): Sets the component at the specified index of this vector to be the specified object. The previous component at that position is discarded.

- l) public void setElementAt(E obj,int index): Sets the component at the specified index of this vector to be the specified object. The previous component at that position is discarded.
- m) public void insertElementAt(E obj,int index): Inserts the specified object as a component in this vector at the specified index.
- n) public E get(int index): Returns the element at the specified position in this Vector.
- o) public E set(int index,E element): Replaces the element at the specified position in this Vector with the specified element.
- p) public boolean add(E e): Appends the specified element to the end of this Vector.
- q) public boolean remove(Object o): Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged. Etc…

**Programs on Vector**:
**Operation**: Check capacity, storing elements, fetching element based on index position, Sorting elements, finding max and min using Collections class method.

```
import java.util.Collections;
import java.util.Vector;

public class VectorOperations {
  public static void main(String[] args) {
    // Checking capacity of Vector
    Vector<String> vector = new Vector<>();
    System.out.println("Initial capacity of Vector: " + vector.capacity());

    // Storing elements in Vector
    vector.add("Java");
    vector.add("C Language");
    vector.add("Python");
    vector.add("Ruby");

    // Displaying elements based on index position
    System.out.println("Elements in the Vector:");
    for (int i = 0; i < vector.size(); i++) {
      System.out.println("Index " + i + ": " + vector.get(i));
    }
    System.out.println();

    // Sorting elements in Vector
    Collections.sort(vector);

    // Displaying sorted elements
    System.out.println("Sorted elements in the Vector:");
    vector.forEach(System.out::println);
    System.out.println();

    // Finding maximum and minimum elements
    String maxElement = Collections.max(vector);
    String minElement = Collections.min(vector);

    System.out.println("Maximum Element: " + maxElement);
    System.out.println("Minimum Element: " + minElement);
  }}
```

**Operation**: Program to prove the ArrayList is better than Vector in Performance.

```
import java.util.*;
public class VectorExampleDemo4
{
        public static void main(String[] args)
        {
                //starting time for Vector
                long startTime = System.currentTimeMillis();

    Vector<Integer> v = new Vector<>();
                 for(int i=0; i<1000000; i++) {
                        v.add(i);
                }
                long endTime = System.currentTimeMillis();
                System.out.println("Time taken by vector :"+(endTime - startTime)+" ms");

                //starting time for ArrayList
                 startTime = System.currentTimeMillis();
                 ArrayList<Integer> al = new ArrayList<>();
                 for(int i=0; i<1000000; i++) {
                        al.add(i);
                }
                 endTime = System.currentTimeMillis();
                System.out.println("Time taken by ArrayList :"+(endTime - startTime)+" ms");
        }
}
```

**Operation**: Adding elements, retrieving elements by using forEach method and lambda expression.

```
import java.util.Vector;
public class VectorExampleDemo5 {
  public static void main(String[] args) {
    Vector<String> names = new Vector<>();
    names.add("Alice");
    names.add("Bob");
    names.add("Charlie");
    names.add("David");

    // Using forEach and Lambda Expression to print each name
    names.forEach(name -> System.out.println(name));
  }
}
```

**Operation**: retrieving all elements by using forEach loop, retrieving elements by using Iterator, retrieving elements by using ListIterator, retrieving elements by using Enumeration and printing the index of specific element.

```
import java.util.Vector;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.Enumeration;


public class VectorOperations {
  public static void main(String[] args) {
    // Creating a Vector with double data
    Vector<Double> vector = new Vector<>();

    // Adding elements to the Vector
    vector.add(2.5);
    vector.add(3.7);
    vector.add(5.1);
    vector.add(4.3);

    // Displaying the elements in the Vector
    System.out.println("Elements in the Vector:");
    for (double element : vector) {
      System.out.println(element);
    }
    System.out.println();

    // Printing the index position of an element
    double searchElement = 5.1;
    int elementIndex = vector.indexOf(searchElement);

    if (elementIndex != -1) {
      System.out.println("Index position of " + searchElement + ": " + elementIndex);
    } else {
      System.out.println(searchElement + " not found in the Vector.");
    }
    System.out.println();

    // Retrieving elements using Iterator
    System.out.println("Retrieving elements using Iterator:");
    Iterator<Double> iterator = vector.iterator();
    while (iterator.hasNext()) {
      System.out.println(iterator.next());
    }
    System.out.println();

    // Retrieving elements using ListIterator
    System.out.println("Retrieving elements using ListIterator (in reverse order):");
    ListIterator<Double> listIterator = vector.listIterator(vector.size());
    while (listIterator.hasPrevious()) {
      System.out.println(listIterator.previous());
    }

    // Retrieving elements using Enumeration
```

```
        System.out.println("Retrieving elements using Enumeration:");
        Enumeration<Double> enumeration = vector.elements();
        while (enumeration.hasMoreElements()) {
            System.out.println(enumeration.nextElement());
        }
    }
}
```

**Stack:**

- public class Stack<E> extends Vector<E>
- It is a predefined class available in java.util package. It is the sub class of Vector class.
- It is a linear data structure that is used to store the Objects in LIFO (Last In first out) order.
- Inserting an element into a Stack is known as push operation whereas extracting an element from the top of the stack is known as pop operation.
- It throws an exception called EmptyStackException, if Stack is empty and we want to fetch the element.
- It has only one constructor as shown below

  Stack s = new Stack();

**Methods of Stack:**

public E push(Object o) :- This method is used to insert an element onto the top of the stack. In Java 1.8, you can use the push method to add elements.

public synchronized E pop() :- To remove and return the element from the top of the Stack

public synchronized E peek() :- Will fetch the element from top of the Stack without removing

public boolean empty() :- Verifies whether the stack is empty or not?

public synchronized int search(Object o) :- It will search a particular element in the Stack and it returns OffSet position. If the element is not present in the Stack it will return -1.

**Programs on Stack:**

**Operations**: adding elements using add method, searching element, retrieving elements by using java 8.

```
import java.util.Stack;

public class StackOperations {
    public static void main(String[] args) {
        // Creating a stack
        Stack<String> stack = new Stack<>();

        // Adding elements to the stack using add method
        stack.add("Banana");
        stack.add("Mango");
        stack.add("Orange");
        stack.add("Grapes");

        // Displaying the elements in the stack
        System.out.println("Elements in the Stack:");
        stack.forEach(System.out::println);
        System.out.println();

        // Searching for an element in the stack
        String searchElement = "Orange";
        int elementIndex = stack.search(searchElement);

        if (elementIndex != -1) {
            System.out.println(searchElement + " found at position " + elementIndex + " from the top of
the stack.");
        } else {
            System.out.println(searchElement + " not found in the stack.");
        }
        System.out.println();

        // Retrieving and displaying elements from the stack
        System.out.println("Retrieving elements from the Stack:");

        while (!stack.isEmpty()) {
            String poppedElement = stack.pop();
            System.out.println("Popped Element: " + poppedElement);
        }
    }
}
```

**Operation**: adding elements on the top, removing and printing the removed element, fetching the top element, checking the stack is empty or not.

```
import java.util.Stack;

public class StackOperations {
    public static void main(String[] args) {
        // Creating a stack
        Stack<String> stack = new Stack<>();

        // Adding elements to the top of the stack
        stack.push("Ravi");
        stack.push("Shyam");
        stack.push("Rahul");
        stack.push("Ruby");

        // Displaying the elements in the stack
        System.out.println("Elements in the Stack:");
        stack.forEach(System.out::println);
        System.out.println();

        // Removing and printing the top element
        if (!stack.isEmpty()) {
            String removedElement = stack.pop();
            System.out.println("Removed Element from the top: " + removedElement);
        } else {
            System.out.println("Stack is empty.");
        }
        System.out.println();

        // Fetching the top element
        if (!stack.isEmpty()) {
            String topElement = stack.peek();
            System.out.println("Top Element of the Stack: " + topElement);
        } else {
            System.out.println("Stack is empty.");
        }
        System.out.println();

        // Checking if the stack is empty or not
        System.out.println("Is the Stack empty? " + stack.isEmpty());
    }
}
```

Difference between ArrayList and Vector.

| ArrayList | Vector |
|---|---|
| An Array List is not synchronized. | Vectors are synchronized, making its thread-safe. |
| As it is not synchronized, ArrayList is fast. | Vector is slow compared to ArrayList, as it is synchronized. |
| Multiple threads are allowed in ArrayList. | Only single thread is allowed in Vector. |
| The performance of Array List is high. | The performance of Vector is low compared to ArrayList. |
| Array List is not a legacy class. | Vector is a legacy as class introduced earlier in Java's history. |
| Array List only uses iterators for traversing. | Vectors can use iterators and enumerations for traversing. |
| Array List increase 50% of its current size in cases the number of elements increases its capacity. | Vector grows 100% of its current size if the number of elements outgrows its capacity. |

**Set Interface:**
public interface **Set<E>** extends Collection<E>

- The Set interface is part of the Java Collections Framework and is a member of the Java Util package (java.util) since 1.2 .
- It extends from Collection Interface and It is a unordered collection.

- It is a collection that contains no duplicate elements.
- At most one null element is allowed.
- Set interface has one sub interface that is SortedSet(java 2v) and It has another sub interface that is NavigableSet(java 6v).
- All known implemented classes are HashSet, EnumSet, LinkedHashSet, TreeSet etc….

**Methods of Set Interface:**

a) int size(): Returns the number of elements in this set.
b) boolean isEmpty(): Returns true if this set contains no elements.
c) boolean contains(Object o): Returns true if this set contains the specified element.
d) boolean add(E e): Adds the specified element to this set if it is not already present (optional operation).
e) boolean remove(Object o): Removes the specified element from this set if it is present (optional operation).
f) boolean containsAll(Collection<?> c): Returns true if this set contains all of the elements of the specified collection.
g) boolean addAll(Collection<? extends E> c): Adds all of the elements in the specified collection to this set if they're not already present .
h) boolean retainAll(Collection<?> c): Retains only the elements in this set that are contained in the specified collection . In other words, removes from this set all of its elements that are not contained in the specified collection.
i) boolean removeAll(Collection<?> c): Removes from this set all of its elements that are contained in the specified collection .
j) void clear(): Removes all of the elements from this set (optional operation). The set will be empty after this call returns.
k) boolean equals(Object o): Compares the specified object with this set for equality. Returns true if the specified object is also a set, the two sets have the same size, and every member of the specified set is contained in this set.

**Behaviour of Set interface Specific classes:**

a. It stores heterogeneous types of data.
b. It stores only unique elements.
c. It stores everything in the form of Object.
d. We can implement generic (<>) to remove the compilation warning and accepting type safe objects.
e. By using generic, we can also take heterogeneous types of elements by using <Object>
f. It does not maintain insertion order and does not have index.
g. We can accept at most one null value not more than that.
h. Set interface uses almost all the methods of Collection interface.

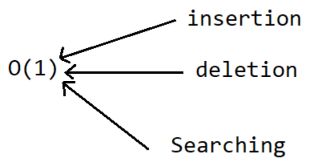
Difference between List and Set:

| List interface | Set interface |
|---|---|
| Lists allow duplicate values | Sets can't have duplicate values |
| Lists can be indexed and positionally accessed | Sets cannot be indexed and cannot be accessed based on their positions |
| Multiple null elements can be stored | Only one null element can be stored at a time |
| List implementations are ArrayList, LinkedList, Vector, Stack. | Set implementations are HashSet, LinkedHashSet. |
| The insertion order is maintained by the List. | It doesn't maintain the insertion order of elements. |
| List is often used when we need to frequently access the elements using their indices. | It is used when we need to store distinct elements. |

| | |
|---|---|
| The listiterator() is used to iterate the List elements. | The iterator() method is used to iterate Set elements. |

**Set Implementer classes:**

What is hashing technique in java ?
-----------------------------------
Hashing is a technique through which we can insert, delete and search an element using O(1).



In hashing technique, internally we are using Hashtable data structure to insert the element.

Hasing technique uses a formula to insert the element object in the Hashtable i.e

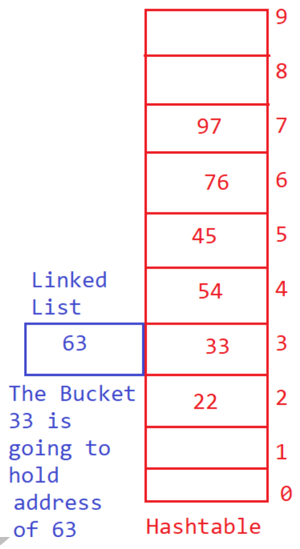
key % 10 [Hash Function]

Let say We want to insert some elements in the Hashtable which are as follows
22,45,54,97,33,76

22 % 10 = 2

Now we want to insert 63 in the Hashtable then
63 % 10 = 3

Here at 3rd bucket location already we have 33 so it is called Hash collision so now the solution is LinkedList



**HashSet** (UNSORTED, UNORDERED, NO DUPLICATES):

public class HashSet exten---ds AbstractSet implements Set, Cloneable, Serializable

- It is a predefined class available in java.util package under Set interface.
- It is an unsorted and unordered set.
- It accepts hetrogeneous kind of data.
- It uses the hashcode of the object being inserted into the Collection. Using this hashcode it finds the bucket location.
- It doesn't contain any duplicate elements as well as It does not maintain any order while iterating the elements from the collection.
- It can accept null value.
- HashSet is used for fast searching operation.

**Constructors of HashSet**:

It contains 4 types of constructors.

1) HashSet hs1 = new HashSet();

It will create the HashSet Object with default capacity is 16. The default load fator or Fill Ratio is 0.75(75% of HashSet is filled up then new HashSet Object will be created having double capacity).

2) HashSet hs2 = new HashSet(int initialCapacity);

will create the HashSet object with user specified capacity.

3) HashSet hs3 = new HashSet(int initialCapacity, float loadFactor);

we can specify our own initialCapacity and loadFactor(by default load factor is 0.75%)

4) HashSet hs = new HashSet(Collection c);

Interconversion of Collection.

**Methods of HashSet:**

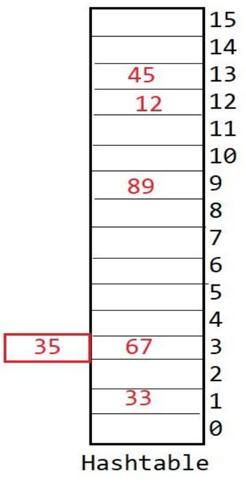
a) public boolean add(E e): Adds the specified element to this set if it is not already present.
b) public boolean remove(Object o): Removes the specified element from this set if it is present.
c) public void clear(): Removes all of the elements from this set. The set will be empty after this call returns.
d) public Object clone(): Returns a shallow copy of this HashSet instance: the elements themselves are not cloned.
e) public boolean contains(Object o): Returns true if this set contains the specified element.
f) public boolean isEmpty(): Returns true if this set contains no elements.
g) public int size(): Returns the number of elements in this set (its cardinality).

**Programs on HashSet**:
**Operation:** The following program describes that HashSet internally uses Hashtable data structure.

```
//Unsorted, Unordered and no duplicates
import java.util.*;
public class HashSetDemo
{
 public static void main(String args[])
 {
        HashSet<Integer>hs = new HashSet<>();
                hs.add(67);
                hs.add(89);
                hs.add(33);
                hs.add(45);
                hs.add(12);
                hs.add(35);

                hs.forEach(str->System.out.println(str));
```

```
            }
}
```

```
How to find the bucket location by using HashSet class
-------------------------------------------------------

HashSet<Integer> hs = new HashSet<>();
      hs.add(67);
      hs.add(89);
      hs.add(33);
      hs.add(45);
      hs.add(12);
      hs.add(35);
Hash Function
key % 16
         67 % 16 = 3

 Output Order is :
 ------------------
 Bottom to top and right to left

 [33  67  35  89  12 45]
```



```
-> It is working with numbers, that means we can find the bucket location
   by using hashing formula

   Ravi % 16

-> Every object is having hashcode
```

$$\boxed{\texttt{hashcode \% 16 = Bucket Location}}$$

```
In the case of HashSet it is 16 because HashSet default
capacity is 16

-> Always the Hashtable will be created based on the capacity.
```

**Operations**: adding elements, retrieving elements, proving that HashSet does not add any duplicate.

```
import java.util.HashSet;

public class HashSetOperations {
  public static void main(String[] args) {
    // Creating a HashSet
    HashSet<String> hashSet = new HashSet<>();

    // Adding elements to the HashSet
    hashSet.add("Banana");
    hashSet.add("Orange");
    hashSet.add("Grapes");
    hashSet.add("Mango");

    // Displaying the elements in the HashSet
    System.out.println("Elements in the HashSet:");
    hashSet.forEach(str -> System.out.println(str));
    System.out.println();
```

```
        // Retrieving elements from the HashSet
        System.out.println("Retrieving elements from the HashSet:");
        for (String element : hashSet) {
            System.out.println(element);
        }
        System.out.println();

        // Adding a duplicate element to the HashSet
        String duplicateElement = "Orange";
        boolean isAdded = hashSet.add(duplicateElement);

        // Displaying the result
        System.out.println("Adding a duplicate element: " + duplicateElement);
        System.out.println("Is the duplicate element added? " + isAdded);
        System.out.println("Elements in the HashSet after attempting to add a duplicate:");
        hashSet.forEach(str -> System.out.println(str));
    }
}
```

**Operations**: Adding all elements from a given Set, removing all the elements from a given Set, retrieving all the elements from a given set.

```
import java.util.HashSet;

public class HashSetOperations {
    public static void main(String[] args) {
        // Creating two HashSets
        HashSet<String> set1 = new HashSet<>();
        HashSet<String> set2 = new HashSet<>();

        // Adding elements to the first HashSet
        set1.add("Ravi");
        set1.add("Ballaya");
        set1.add("Nagarjuna");

        // Displaying elements in the first HashSet
        System.out.println("Elements in the First HashSet:");
        set1.forEach(System.out::println);
        System.out.println();

        // Adding elements to the second HashSet
        set2.add("Ram");
        set2.add("Shayam");
        set2.add("Kalyan");

        // Displaying elements in the second HashSet
        System.out.println("Elements in the Second HashSet:");
        set2.forEach(System.out::println);
        System.out.println();

        // Using addAll to add all elements from the second HashSet to the first HashSet
        set1.addAll(set2);
        // Displaying elements in the first HashSet after using addAll
        System.out.println("Elements in the First HashSet after using addAll:");
        set1.forEach(System.out::println);
        System.out.println();

        // Using removeAll to remove all common elements from the first HashSet
        set1.removeAll(set2);

        // Displaying elements in the first HashSet after using removeAll
        System.out.println("Elements in the First HashSet after using removeAll:");
        set1.forEach(System.out::println);
        System.out.println();

        // Using retainAll to retain only common elements in the first HashSet
        set1.retainAll(set2);

        // Displaying elements in the first HashSet after using retainAll
        System.out.println("Elements in the First HashSet after using retainAll:");
        set1.forEach(System.out::println);
    }
}
```

**Operation**: prove that Collection interface add method returns boolean and check the given element present or not present.

```
import java.util.*;
public class HashSetDemo2
{
        public static void main(String[] args) {
                boolean[] ba = new boolean[6];
                Set s = new HashSet();
                 ba[0] = s.add("a");
                ba[1] = s.add(42);
                ba[2] = s.add("b");
                ba[3] = s.add("a");
                ba[4] = s.add("new Object()");
                ba[5] = s.add(new Object());
        for(int x = 0; x<ba.length; x++)
          System.out.println(ba[x]+"   ");

                if(s.contains(42)) { //Searching a particular object in the Set
                                System.out.println("Object 42 is available ...");
                        }
                        else {
                                System.out.println("42 is not available ...");
                        }


  System.out.println("Fetching the elements of HashSet");
                        s.forEach(str -> System.out.println(str));
    }
}
```

**Linked HashSet:**

- public class LinkedHashSet extends HashSet implements Set, Cloneable, Serializable.
- It is a predefined class in java.util package under Set interface since 1.4v
- It is the sub class of HashSet class.
- It is an ordered version of HashSet that maintains a doubly linked list across all the elements.
- We should use LinkedHashSet class when we want to store unique objects in insertion an order.
- When we iterate the elements through HashSet the order will be unpredictable, while when we iterate the elements through Linked Hash Setsthen the order will be same as they were inserted in the collection.
- It accepts heterogeneous and null value is allowed but only one.
- It has same constructor as HashSet class.
- We do not have any special methods in this class HashSet. We must use the same methods those are inherited and implementing from Set and Collection interfaces.

**Programs on LinkedHashSet:**

**Operations**: adding elements, checking the size, searching elements, fetching elements, remove an element, remove all elements.

```
import java.util.LinkedHashSet;
public class LinkedHashSetDemo1 {
   public static void main(String[] args) {
      LinkedHashSet<Integer> linkedHashSet = new LinkedHashSet<>();
       linkedHashSet.add(10);
       linkedHashSet.add(5);
       linkedHashSet.add(15);
       linkedHashSet.add(20);
       linkedHashSet.add(5);

       linkedHashSet.forEach(x -> System.out.println(x));
       System.out.println("LinkedHashSet size: " + linkedHashSet.size());

       int elementToCheck = 15;
       if (linkedHashSet.contains(elementToCheck))  {
          System.out.println(elementToCheck + " is present in the LinkedHashSet.");
       } else  {
          System.out.println(elementToCheck + " is not present in the LinkedHashSet.");
       }

       int elementToRemove = 10;
       linkedHashSet.remove(elementToRemove);
       System.out.println("After removing " + elementToRemove + ", LinkedHashSet elements: " +
linkedHashSet);

            linkedHashSet.clear();
       System.out.println("After clearing, LinkedHashSet elements: " + linkedHashSet);
   }
```

}

**SortedSet Interface:**

public interface SortedSet<E> extends Set<E>
1) It is the sub interface of Set interface since 1.2v.

2) If we don't want duplicate elements and want to store the elements based on some sorting order i.e default natural sorting order then we should go with SortedSet(I).

3) default natural sorting means, if it is number then ascending order and if it is String then dictionary order or Alphabetical order.

4) We have two interfaces Comparable(available in java.lang package) and Comparator (available in java.util package) to compare two objects.

**Methods of SortedSet:**

a) Comparator<? super E> comparator(): Returns the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements.
b) SortedSet<E> subSet(E fromElement, E toElement): Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
c) SortedSet<E> headSet(E toElement): Returns a view of the portion of this set whose elements are strictly less than toElement.
d) SortedSet<E> tailSet(E fromElement): Returns a view of the portion of this set whose elements are greater than or equal to fromElement.
e) E first()L Returns the first (lowest) element currently in this set.
f) E last():Returns the last (highest) element currently in this set.
g) default Spliterator<E> spliterator(): Creates a Spliterator over the elements in this sorted set.

**Comparable and Comparator:**

**What is the difference between Comparable and Comparator interface?**
1. Comparable is used for providing natural sorting order.
2. Comparator is used for providing custom sorting order.
3. 2.Comparable contains compareTo(Object) param method providing objects comparison logic.
4. Comparator contains compare(Object, Object) param method for providing objects comparison logic.
5. Comparable is available in java.lang package since Java 1.2v.
6. Comparator is available in java.util package since Java 1.2v.
7. Both interfaces are functional interfaces.
8. Comparable interface has only one method that is compareTo(-).It does not have any additional Java 8v default or static or private methods.
9. But in Comparator interface, in addition to compare(-, -) method.It has another abstract method equals(Object) and several Java 8v default and static methods.

**Program On Comparable(I):**
<u>Employee.java</u>
```
public class Employee implements Comparable<Employee> {
  private Integer employeeId;
  private String employeeName;
  private Double employeeSalary;
```

```
 public Employee(Integer employeeId, String employeeName, Double employeeSalary)  {
                    super();
                    this.employeeId = employeeId;
                    this.employeeName = employeeName;
                    this.employeeSalary = employeeSalary;
  }
  @Override
   public String toString() {
           return "Employee [employeeId=" + employeeId + ", employeeName=" + employeeName + ",
employeeSalary="+ employeeSalary + "]";
    }
//Sorting based on the employeeId
 @Override
 public int compareTo(Employee emp) {
          return this.employeeId - emp.employeeId;
   }
}
```

ComparableDemo.java

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

public class ComparableDemo {
          public static void main(String[] args)
          {
                    ArrayList<Employee> empData = new ArrayList<>();
                    empData.add(new Employee(104, "Aryan", 12890.89));
                    empData.add(new Employee(103, "Zuber", 14890.89));
                    empData.add(new Employee(101,  "Satish",11890.89));
                    empData.add(new Employee(102, "Raj", 14890.89));

                    Collections.sort(empData);
                    empData.forEach(emp-> System.out.println(emp));
          }
}
```

**Limitation of Comparable interface:**

1) We need to modify the original source code (BLC class), If the source code is not available then it is not possible to perform  sorting operation.
2) We can provide only one sorting logic if we want to provide mutiple sorting logic then it is not possible.

To avoid the above said problems we introduced Comparator interface available in java.util package.

**Program on Comparator interface(I):**

Product.java

```
public class Product {
  private Integer productId;
  private String productName;
  private Double productPrice;

public Product(Integer productId, String productName, Double productPrice) {
	super();
	this.productId = productId;
	this.productName = productName;
	this.productPrice = productPrice;
}
public Integer getProductId() {
	return productId;
}
public void setProductId(Integer productId) {
	this.productId = productId;
}
public String getProductName() {
	return productName;
}
public void setProductName(String productName) {
	this.productName = productName;
}
public Double getProductPrice() {
	return productPrice;
}
public void setProductPrice(Double productPrice) {
	this.productPrice = productPrice;
}
@Override
public String toString() {
	return "Product [productId=" + productId + ", productName=" + productName + ",
productPrice=" + productPrice + "]";
}

}
```

ProductComparator.java

---------------

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

public class ProductComparator {

	public static void main(String[] args) {
		ArrayList<Product> listOfProduct = new ArrayList<>();
		listOfProduct.add(new Product(3, "Camera", 16450.89));
		listOfProduct.add(new Product(1, "Laptop", 76450.89));
		listOfProduct.add(new Product(2, "Mobile", 22450.89));
		//Sorting based on Product Id
		Comparator<Product> prodId = new Comparator<Product>() {
			@Override
			public int compare(Product p1, Product p2) {
				return p1.getProductId()-p2.getProductId();
			}
		};


		Collections.sort(listOfProduct, prodId);
		System.out.println("Sorting based on the Product ID");

		listOfProduct.forEach(p -> System.out.println(p));

		//Sorting based on Product Name

		Comparator<Product> prodName = (p1, p2)->
p1.getProductName().compareTo(p2.getProductName());

		Collections.sort(listOfProduct, prodName);
		System.out.println("Sorting based on the Product Name");

		listOfProduct.forEach(p -> System.out.println(p));

	}
}
```

**Program to sort Integer object data in descending order**:

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

public class IntegerComparator {
	public static void main(String[] args) {
		ArrayList<Integer> al = new ArrayList<>();
		al.add(67);
		al.add(57);
		al.add(47);
		al.add(37);
		al.add(17);
		Comparator<Integer> descSort = (i1, i2)-> -(i1-i2);
		Collections.sort(al,descSort);
		al.forEach(x -> System.out.println(x));
	}
}
```

**[Note: By default TreeSet add(-) invokes Comparable.compareTo(-) method, when we are adding objects in TreeSet by using add(-) methods invokes Comparator.compare(O, O) onlywhen we pass it as argument explicitly by using TreeSet(Comparator) constructor.]**

Difference between Comparable and Comparator

| Comparable | Comparator |
|---|---|
| Comparable is an interface in Java. | Comparator is a functional interface in Java. |
| Comparable provides compareTo() method to sort elements in Java. | Comparator provides compare() method to sort elements in Java. |
| Comparable interface is present in java.lang package. | Comparator interface is present in java.util package. |
| It provides single sorting sequences. Ex: Sort either by id or name | It provides multiple sorting sequences. Ex. Sort by both id and name. |
| Comparable can be used for natural or default ordering. | Comparator can be used for custom ordering. |
| Comparable modifies the class that implements it. | Comparator doesn't modify any class. |
| It uses sort(List l). | It uses sort(List l, Comparator t). |

**TreeSet:**

- public class TreeSet<E> extends AbstractSet<E> implements NavigableSet, Cloneable, Serializable
- It is a predefined class available in java.util package under Set interface since 1.2v..
- TreeSet, TreeMap and PriorityQueue are the three sorted collection in the entire Collection Framework so these classes never accepting heterogeneous kind of the data.
- It will sort the elements in natural sorting order i.e ascending order in case of number, and alphabetical order or Dictionary order in the case of String. In order to sort the elements, It uses Comparator interface.
- It does not accept duplicate and null value(java.lang.NullPointerException).
- It does not accept heterogeneous type of data if we try to insert it will throw a runtime exception i.e java.lang.ClassCastException.
- TreeSet implements NavigableSet and NavigableSet extends SortedSet.
- Note that this implementation is not synchronized.

**Constructors of TreeSet:**

It contains 4 types of constructors:

1) TreeSet t1 = new TreeSet();

Creates an empty TreeSet object, elements will be inserted in a natural sorting order.

2) TreeSet t2 = new TreeSet(Comparator c);

Constructs a new, empty tree set, sorted according to the specified comparator.

3)TreeSet t3 = new TreeSet(Collection c);

Constructs a new tree set containing the elements in the specified collection, sorted according to the *natural ordering* of its elements.

4)TreeSet t4 = new TreeSet(SortedSet s);

Constructs a new tree set containing the same elements and using the same ordering as the specified sorted set.

**Methods of TreeSet:**

a) public Iterator<E> iterator(): Returns an iterator over the elements in this set in ascending order.
b) public Iterator<E> descendingIterator()(1.6v): Returns an iterator over the elements in this set in descending ordfeer.
c) public NavigableSet<E> descendingSet()(1.6v): Returns a reverse order view of the elements contained in this set.
d) public int size(): Returns the number of elements in this set .
e) public boolean isEmpty(): Returns true if this set contains no elements.
f) public boolean contains(Object o): Returns true if this set contains the specified element.
g) public boolean add(E e Adds the specified element to this set if it is not already present.
h) public boolean remove(Object o): Removes the specified element from this set if it is present.NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive): Returns a view of the portion of this set whose elements range from from Element to to Element.
i) NavigableSet<E> headSet(E toElement, boolean inclusive): Returns a view of the portion of this set whose elements are less than (or equal to, if inclusive is true) to Element.
j) NavigableSet<E> tailSet(E fromElement, boolean inclusive): Returns a view of the portion of this set whose elements are greater than (or equal to, if inclusive is true) fromElement.
Etc…………

**Programs on TreeSet:**

**Operations**: adding elements, printing in ascending and descending order, fetching elements by using for Each Remaing().

```
import java.util.*;
public class TreeSetDemo1 {
        public static void main(String[] args)  {
                SortedSet<String> t1 = new TreeSet<>();
                t1.add("Orange");
                t1.add("Mango");
                t1.add("Pear");
                t1.add("Banana");
                t1.add("Apple");
                System.out.println("In Ascending order");
                t1.forEach(i -> System.out.println(i));

                TreeSet<String> t2 = new TreeSet<>();
                t2.add("Orange");
                t2.add("Mango");
                t2.add("Pear");
                t2.add("Banana");
                t2.add("Apple");

    System.out.println("In Descending order");
                Iterator<String> itr2 = t2.descendingIterator();  //for descending order
    //From 1.8 to replace hasNext() and next() method
     itr2.forEachRemaining(x -> System.out.println(x));
```

```
            }
}
```

**[Note: - descendingIterator() is a predefined method of TreeSet class which will traverse in the descending order.]**

**Operations**: Working with customized sorting order using Comparator as a parameter.

```
import java.util.TreeSet;

public class Tree_Set {
            public static void main(String[] args) {
                        System.out.println("Sorting name -> Ascending Order");
                        TreeSet<Employee> ts1 = new TreeSet<>((e1,e2)-
>e1.getName().compareTo(e2.getName()));

                        ts1.add(new Employee(101, "Omm", 24));
                        ts1.add(new Employee(201, "Appu", 27));
                        ts1.add(new Employee(301, "Juju", 26));
                        ts1.forEach(i -> System.out.println(i));

      System.out.println("____________________________________________");
                        System.out.println("Sorting name -> Descending Order");

                        TreeSet<Employee> ts2= new TreeSet<>((e1,e2)-
>e2.getName().compareTo(e1.getName()));
                        ts2.add(new Employee(101, "Omm", 24));
                        ts2.add(new Employee(201, "Appu", 27));
                        ts2.add(new Employee(301, "Juju", 26));
      ts2.forEach(i -> System.out.println(i));

      System.out.println("____________________________________________");
                        System.out.println("Sorting Age -> Ascending Order");

                        TreeSet<Employee> ts3= new TreeSet<>((e1,e2)-
>e1.getAge().compareTo(e2.getAge()));
                        ts3.add(new Employee(101, "Omm", 24));
                        ts3.add(new Employee(201, "Appu", 27));
                        ts3.add(new Employee(301, "Juju", 26));
      ts3.forEach(i -> System.out.println(i));

                        System.out.println("____________________________________________");
                        System.out.println("Sorting Age ->  Descending Order");

                        TreeSet<Employee> ts4= new TreeSet<>((e1,e2)-
>e2.getAge().compareTo(e1.getAge()));
                        ts4.add(new Employee(101, "Omm", 24));
                        ts4.add(new Employee(201, "Appu", 27));
                        ts4.add(new Employee(301, "Juju", 26));
      ts4.forEach(i -> System.out.println(i));

      System.out.println("____________________________________________");
                        System.out.println("Sorting Id -> Ascending Order");
```

```
                TreeSet<Employee> ts5= new TreeSet<>((e1,e2)->e1.getId().compareTo(e2.getId()));
                ts5.add(new Employee(101, "Omm", 24));
                ts5.add(new Employee(201, "Appu", 27));
                ts5.add(new Employee(301, "Juju", 26));
        ts5.forEach(i -> System.out.println(i));

        System.out.println("................................................");
        System.out.println("Sorting Id ->  Descending Order");

        TreeSet<Employee> ts6= new TreeSet<>((e1,e2)->e2.getId().compareTo(e1.getId()));
        ts6.add(new Employee(101, "Omm", 24));
        ts6.add(new Employee(201, "Appu", 27));
        ts6.add(new Employee(301, "Juju", 26));
        ts6.forEach(i -> System.out.println(i));
            }
}
class Employee {
        Integer id;
        String name;
        Integer age;

        public Employee(int id, String name, int age){
                this.id = id;
                this.name = name;
                this.age = age;
        }

        public Integer getId() {
                return id;
        }

        public String getName() {
                return name;
        }

        public Integer getAge() {
                return age;
        }

        @Override
        public String toString(){
                return " " + this.id + " " + this.name + " "+ this.age;
        }
}
```

**Sorted Set interface:**

public interface SortedSet<E>extends Set<E>

- A Set that further provides a *total ordering* on its elements.
- The elements are ordered using their natural ordering, or by a Comparator typically provided at sorted set creation time.
- The set's iterator will traverse the set in ascending element order.

- TreeSet is a Implemented class of SortedSet.

**Methods of SortedSet interface:**

a) public E first() :- Will fetch first element.
b) public E last() :- Will fetch last element.
c) public SortedSet headSet(int range) :- Will fetch the values which are less than specified range.
d) public SortedSet tailSet(int range) :- Will fetch the values which are equal or greater than the specified range.
e) public SortedSet subSet(int startRange, int endRange) :- Will fetch the range of values where startRange is inclusive and endRange is exclusive.

**Programs on SortedSet:**

**Operation**: adding elements, fetching first and last element, fetching values less than a range, fetching elements equal or greater than specific range, fetching values from starting range to ending range.

```
import java.util.*;
public class SortedSetMethodDemo
{
      public static void main(String[] args)  {
          TreeSet<Integer> times = new TreeSet<>();
          times.add(1205);
          times.add(1505);
          times.add(1545);
          times.add(1600);
          times.add(1830);
          times.add(2010);
          times.add(2100);

          SortedSet<Integer> sub = new TreeSet<>();
                              sub =  times.subSet(1545,2100);
          System.out.println("Using subSet() :-"+sub);//[1545, 1600,1830,2010]
          System.out.println(sub.first());
          System.out.println(sub.last());
                     sub = times.headSet(1545);
                              System.out.println("Using headSet() :-"+sub); //[1205, 1505]
                      sub = times.tailSet(1545);
                              System.out.println("Using tailSet() :-"+sub); //[1545 to 2100]
      }
}
```

**Operations**: Develop a program to store and fetch the elements by using TreeSet, SortedSet, NavigableSet.

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

public class TreeSetDemo {
    public static void main(String[] args) {
        // Example of a SortedSet
        SortedSet<Integer> t1 = new TreeSet<>();
        t1.add(4);
        t1.add(7);
        t1.add(2);
        t1.add(1);
        t1.add(9);

        System.out.println("SortedSet (Natural Order): " + t1);

        // Example of a NavigableSet
        NavigableSet<String> t2 = new TreeSet<>();
        t2.add("Orange");
        t2.add("Mango");
        t2.add("Banana");
        t2.add("Grapes");
        t2.add("Apple");

        System.out.println("NavigableSet: " + t2);
    }
}
```

**NavigableSet:**

public interface NavigableSet<E> extends SortedSet<E>

- NavigableSet is a sub-interface of the SortedSet interface.
- It extends SortedSet to provide navigation methods for retrieving elements based on their relationship to other elements in the set.
- With the help of SortedSet interface method we can find out the range of values but we can't navigate among those elements.
- Now to frequently navigate among those range of elements, Java software people introduced new interface called NavigableSet from 1.6V

**Methods of NavigableSet:**

a) E lower(E e): Returns the greatest element in this set strictly less than the given element, or null if there is no such element.
b) E floor(E e): Returns the greatest element in this set less than or equal to the given element, or null if there is no such element.
c) E ceiling(E e): Returns the least element in this set greater than or equal to the given element, or null if there is no such element.
d) E higher(E e): Returns the least element in this set strictly greater than the given element, or null if there is no such element.
e) E pollFirst(): Retrieves and removes the first (lowest) element, or returns null if this set is empty.
f) E pollLast(): Retrieves and removes the last (highest) element, or returns null if this set is empty.
g) NavigableSet<E> descendingSet(): Returns a reverse order view of the elements contained in this set. The descending set is backed by this set, so changes to the set are reflected in the descending set, and vice-versa

**Programs on NavigableSet:**
**Operations**: Use of lower, floor, ceiling, higher methods of NavigableSet.

```
import java.util.*;

public class NavigableSetDemo
{
   public static void main(String[] args)
   {
      NavigableSet<Integer> ns = new TreeSet<>();
      ns.add(1);
      ns.add(2);
      ns.add(3);
      ns.add(4);
      ns.add(5);
      ns.add(6);

                        System.out.println("lower(3): " + ns.lower(3));//Just below than the specified
element or null

      System.out.println("floor(3): " + ns.floor(3)); //Equal or less or null

      System.out.println("higher(3): " + ns.higher(3));//Just greater than specified element or null

     System.out.println("ceiling(3): " + ns.ceiling(3));//Equal or greater or null

   }
}
```

**Operation**: Write a program to show the uses of NavigableSet methods.

```
import java.util.NavigableSet;

import java.util.TreeSet;

public class NavigableSetExample {
   public static void main(String[] args) {
      NavigableSet<Integer> navigableSet = new TreeSet<>();

      // Adding elements to the NavigableSet
      navigableSet.add(10);
      navigableSet.add(20);
      navigableSet.add(30);
      navigableSet.add(40);
      navigableSet.add(50);

      // Displaying the NavigableSet
      System.out.println("NavigableSet: " + navigableSet);

      // Using NavigableSet methods
      System.out.println("Ceiling for 35: " + navigableSet.ceiling(35));
```

```
        System.out.println("Higher than 25: " + navigableSet.higher(25));
        System.out.println("Floor for 35: " + navigableSet.floor(35));
        System.out.println("Lower than 25: " + navigableSet.lower(25));

        // Polling the first and last elements
        System.out.println("Polling First: " + navigableSet.pollFirst());
        System.out.println("Polling Last: " + navigableSet.pollLast());

        // Displaying the NavigableSet after polling
        System.out.println("NavigableSet after polling: " + navigableSet);
    }
}
```

***Explanation*:**
We create a NavigableSet using TreeSet.

The ceiling method returns the least element greater than or equal to the given element (35 in this case).

The higher method returns the least element strictly greater than the given element (25 in this case).

The floor method returns the greatest element less than or equal to the given element (35 in this case).

The lower method returns the greatest element strictly less than the given element (25 in this case).

pollFirst and pollLast methods remove and return the first and last elements, respectively.

**Operations**: Working with NavigableSet for Range Operation.

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

public class NavigableSetRangeExample {
    public static void main(String[] args) {
        NavigableSet<Integer> navigableSet = new TreeSet<>();

        // Adding elements to the NavigableSet
        for (int i = 1; i <= 10; i++) {
            navigableSet.add(i * 10);
        }

        // Displaying the NavigableSet
        System.out.println("NavigableSet: " + navigableSet);

        // Subsets and Range Operations
        NavigableSet<Integer> subset = navigableSet.subSet(30, true, 70, true);
        System.out.println("Subset (30 to 70): " + subset);

        NavigableSet<Integer> descendingSubset = navigableSet.descendingSet().subSet(70, true, 30,
true);
        System.out.println("Descending Subset (70 to 30): " + descendingSubset);
    }
}
```

***Explanation:***

We create a NavigableSet with elements 10, 20, ..., 100.

The subSet method creates a subset including the starting element (30) and ending element (70).

The descendingSet method returns a reverse-order view of the set.

We then create a subset of the descending set using subSet, resulting in a descending subset from 70 to 30.

These examples demonstrate the power of NavigableSet in performing range-based operations and efficiently retrieving elements based on their relationships.

**Operations**: Descending Iterator with NavigableSet.

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

public class DescendingIteratorExample {
  public static void main(String[] args) {
    NavigableSet<String> navigableSet = new TreeSet<>();

    // Adding elements to the NavigableSet
    navigableSet.add("Apple");
    navigableSet.add("Banana");
    navigableSet.add("Grapes");
    navigableSet.add("Mango");
    navigableSet.add("Orange");

    // Using descendingIterator to iterate in reverse order
    Iterator<String> descendingIterator = navigableSet.descendingIterator();

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

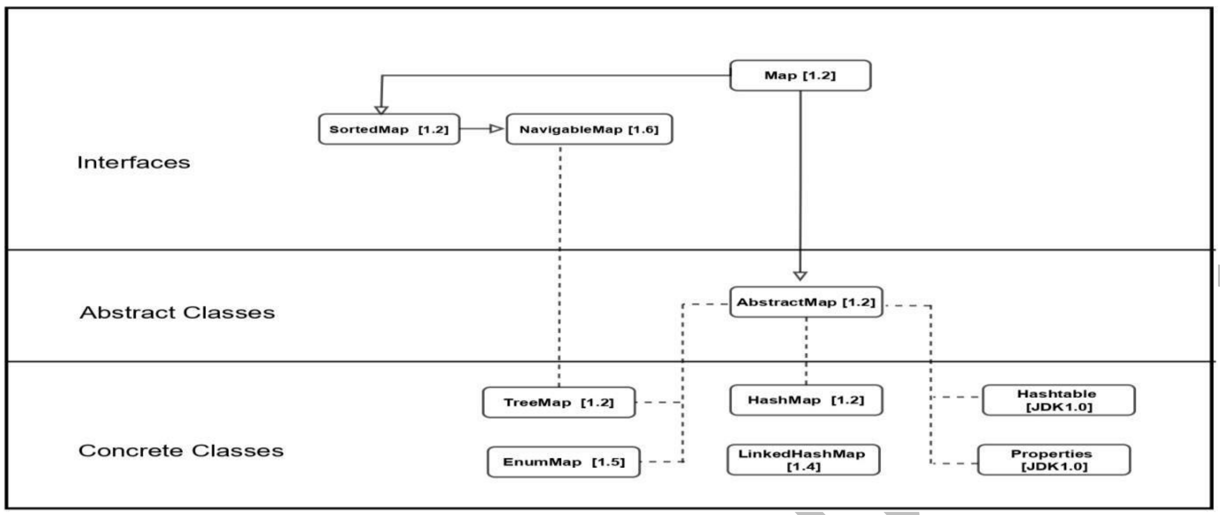
***Explanation***:

The descendingIterator method returns an iterator over the elements in reverse order.

We use this iterator to traverse and print elements in reverse order.

These additional examples showcase the versatility of the NavigableSet interface and its practical applications in real-world scenarios.

**Map interface:**
**Map interface Hierarchy:**



Interface Map<K,V>

- Type Parameters:
  K - the type of keys maintained by this map

  V - the type of mapped values
- As we know Collection interface is used to hold single or individual object but Map interface will hold group of objects in the form key and value pair.
- Map interface is not the part of Collection.
- Before Map interface We had Dictionary(abstract class) class and it is implemented Hashtable class in JDK 1.0V.
- Map interface works with key and value pair introduced from 1.2V.
- Here key and value both are objects.
- Here key must be unique and value may be duplicate.
- Each key and value pair is creating one Entry(Entry is nothing but the combination of key and value pair).

```
interface Map{
    interface Entry {
     }
}
```

- The Map interface provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings.
- Map interface has one sub interface that is SortedMap and its has another sub interface that is NavigableMap.

**Methods of Map interface:**

1) Object put(Object key, Object value) :- To insert one entry in the Map collection. It will return the old object value if the key is already available(Duplicate key).

2) void putAll(Map m) :- Merging of two Map collection
3) int size() :- To count the pair of key and value or Entry
4) void clear() :- Used to clear the Map
5) boolean isEmpty() :- To verify Map is empty or not?
6) boolean containsKey(Object key): Returns true if this map contains a mapping for the specified key.
7) boolean containsValue(Object value): Returns true if this map maps one or more keys to the specified value.
8) Object get(Object key) :- It will return corresponding value of key, if the key is not present then it will return null.
9) Object getOrDefault(Object key, Object defaultValue) :- To avoid null value this method has been introduced, here we can pass some defaultValue to avoid the null value.
10) remove(Object key) :- One complete entry will be removed.
11) putIfAbsent(Object key, Object value) :- It will insert an entry if and only if key is not present , if the key is already available then it will not insert the Entry to the Map Collection.
12) int size(): Returns the number of key-value mappings in this map.
13) default boolean remove(Object key, Object value): Removes the entry for the specified key only if it is currently mapped to the specified value.
14) default boolean replace(K key, V oldValue, V newValue): Replaces the entry for the specified key only if currently mapped to the specified value.
15) default V replace(K key, V value): Replaces the entry for the specified key only if it is currently mapped to some value.

**Collection views Methods:**

1) Set<K> keySet(): Returns a Set view of the keys contained in this map.
2) Collection<V> values(): Returns a Collection view of the values contained in this map.
3) Set<Map.Entry<K,V>> entrySet(): Returns a Set view of the mappings contained in this map.
4)

**Map implementer classes:**
**HashMap**:[Unsorted, Unordered, No Duplicate keys]
public class HashMap<K,V>extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable

Type Parameters:
K - the type of keys maintained by this map

V - the type of mapped values

- HashMap is Hash-tablebased implementation of the Map interface.
- It gives us unsorted and Unordered map. when we need a map and we don't care about the order while iterating the elements through it then we should use HashMap.
- It inserts the element based on the hashCode of the Object key using hashing technique [hashing algorithm]
- It does not accept duplicate keys but value may be duplicate.
- It accepts only one null key(because duplicate keys are not allowed) but multiple null values are allowed.
- Time complexity of search, insert and delete will be O(1)

- We should use HashMap to perform searching operation.
- This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.
- The default load factor of HashTable is 0.75.
- The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased.
- We must choose HashMap<K,V> to store only unique entries in single thread application.

**Constructors of HashMap:**

1) HashMap()

Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).

2) HashMap(int initialCapacity)

Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).

3) HashMap(int initialCapacity, float loadFactor)

Constructs an empty HashMap with the specified initial capacity and load factor.

4) HashMap(Map<? extends K,? extends V> m)

Constructs a new HashMap with the same mappings as the specified Map.

**Methods of HashMap:** There is no extra method available in HashMap; all the methods are either overridden from AbstractMap or implemented from the Map interface.So check the Map interface methods description.

**How HashMap works internally:**

a) While working with HashSet or HashMap every object must be compared because duplicate objects are not allowed.

b) Whenever we add any new key to verify whether key is unique or duplicate, HashMap internally uses hashCode(), == operator and equals method.

c) While adding the key object in the HashMap, first of all it will invoke the hashCode() method to retrieve the corresponding key hashcode value.

Example :- hm.put(key,value);

then internally key.hashCode();

d) If the newly added key and existing key hashCode value both are same (Hash collision), then only == operator is used for comparing those keys by using reference or memory address, if both keys references are same then existing key value will be replaced with new key value.

- If the reference of both keys are different then equals(Object obj) method is invoked to compare those keys by using state(data).

- If the equals(Object obj) method returns true (content wise both keys are same), this new key is duplicate then existing key value will be replaced.
- If equals(Object obj) method returns false, this new key is unique, new entry (key-value) will be inserted.

  ***Note**: equals(Object obj) method is invoked only when two keys are having same hashcode as well as their references are different.*

e) Actually by calling hashcode method we are not comparing the objects, we are just storing the objects in a group so the currently adding key object will be compared with its HASHCODE GROUP objects, but not with all the keys which are available in the Map.

f) The main purpose of storing objects into the corresponding group to decrease the number of comparison so the efficiency of the program will increase.

g) To insert an entry in the HashMap, HashMap internally uses Hashtable data structure.

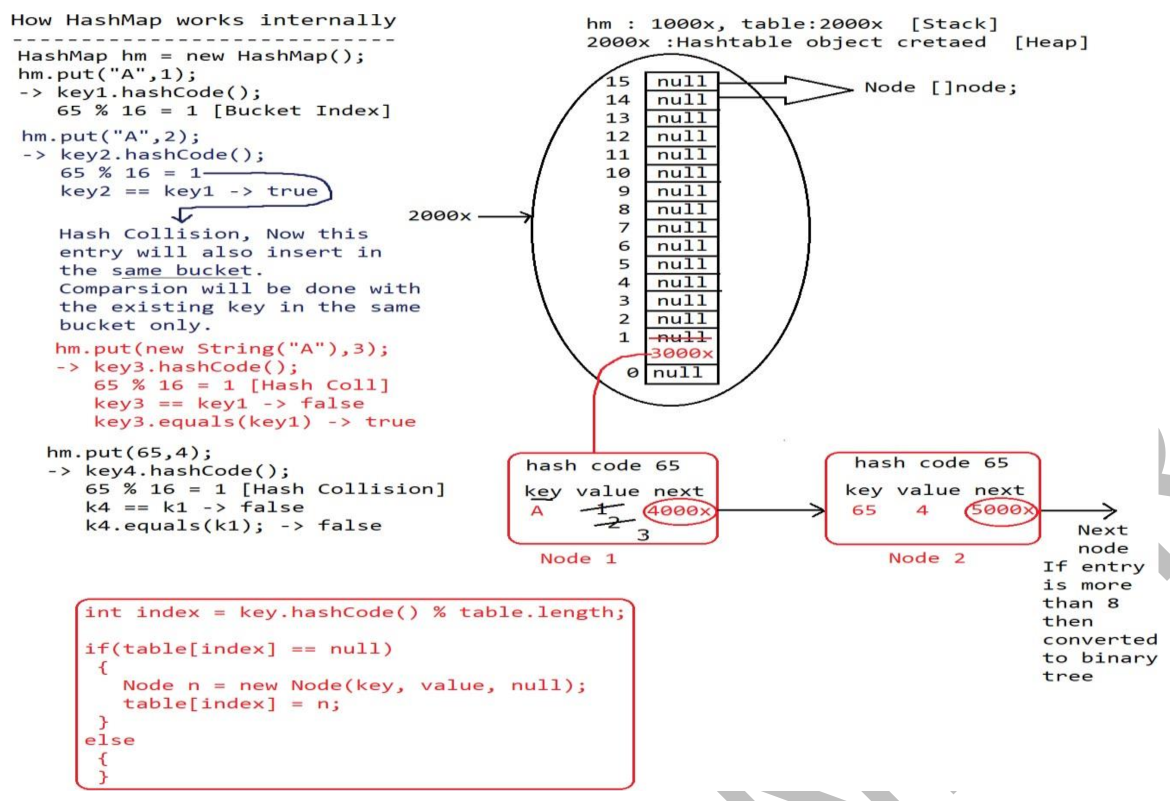
h) Now, for storing same hashcode object into a single group, hash table data structure internally uses one more data structure called Bucket.

i) The Hash table data structure internally uses Node class array object.

j) The bucket data structure internally uses LinkedList data structure, It is a single linked list again implemented by Node class only.

K) A bucket is group of entries of same hash code keys.

L) Performance wise LinkedList is not good to search, so from java 8 onwards LinkedList is changed to Binary tree to decrease the number of comparison within the same bucket hashcode if the number of entries are greater than 8.



```
How HashMap works internally
-----------------------------
 HashMap hm = new HashMap();
 hm.put("A",1);
 -> key1.hashCode();
    65 % 16 = 1 [Bucket Index]
 hm.put("A",2);
 -> key2.hashCode();
    65 % 16 = 1
    key2 == key1 -> true

    Hash Collision, Now this
    entry will also insert in
    the same bucket.
    Comparsion will be done with
    the existing key in the same
    bucket only.
 hm.put(new String("A"),3);
 -> key3.hashCode();
    65 % 16 = 1 [Hash Coll]
    key3 == key1 -> false
    key3.equals(key1) -> true
 hm.put(65,4);
 -> key4.hashCode();
    65 % 16 = 1 [Hash Collision]
    k4 == k1 -> false
    k4.equals(k1); -> false
```

```
int index = key.hashCode() % table.length;
if(table[index] == null)
 {
   Node n = new Node(key, value, null);
   table[index] = n;
 }
else
 {
 }
```

**equals() and hashCode() method contract :**

If equals() method invoked on two objects and it returns true then hashcode of both the objects must be same.

Example:

```
import java.util.*;
public class HashMapDemo
{
        public static void main(String[] args)
        {
                /*Map<String,String> map = new HashMap<>();
                map.put("Ravi","Ampt");
                map.put(new String("Ravi"),"Hyd");
                System.out.println(map.size());*/

                /*String str1 = "Ravi";
                Object str2 = new String("Ravi");
                System.out.println(str1.hashCode());
                System.out.println(str2.hashCode());*/

                Map<Integer,String> map = new HashMap<>();
                map.put(128,"Ampt");
                map.put(new Integer(128),"Hyd");
                System.out.println(map.size());

                String str = "A";
                System.out.println(str.hashCode());
        }
}
```

**Programs on HashMap:**

**Operations**: Develop a program to create objects of HashMap by using all 4 constructors and insert some entries and display it.

```
import java.util.HashMap;

public class HashMapExample {

  public static void main(String[] args) {

                //Default constructor

        HashMap<String, Integer> hashMap1 = new HashMap<>();
                hashMap1.put("Amar", 101);
                hashMap1.put("Ravi", 107); //Ravi is key and 107 is the value
                hashMap1.put("Virat", 103);
                hashMap1.put("Rahit", 105);
                System.out.println("HashMap 1: " + hashMap1);

                // Constructor with initial capacity

                int initialCapacity = 10;
HashMap<String, Integer> hashMap2 = new HashMap<>(initialCapacity);
                hashMap2.put("Grapes", 50);
                hashMap2.put("Apple", 150);
                hashMap2.put("Mango", 100);
                hashMap2.put("Orange", 80);

                System.out.println("HashMap 2: " + hashMap2);

                // Constructor with initial capacity and load factor

                int customInitialCapacity = 5;
                float loadFactor = 0.75f;
HashMap<String, Integer>hashMap3 = new HashMap<>(customInitialCapacity, loadFactor);
                hashMap3.put("Dog", 5);
                hashMap3.put("Tiger", 8);
                hashMap3.put("Lion", 6);
                hashMap3.put("Cat", 4);

                System.out.println("HashMap 3: " + hashMap3);

                // Constructor with another Map

                HashMap<String, Integer>numberMap = new HashMap<>();
                numberMap.put("One", 1);
                numberMap.put("Two", 2);
                numberMap.put("Three", 3);
```

```
 HashMap<String, Integer> hashMap4 = new HashMap<>(numberMap);
                     System.out.println("HashMap 4: " + hashMap4);
          }
}
```

**Operations**: Program to shows the HashMap is unordered.

```
import java.util.*;
public class HashMapDemo {
    public static void main(String[] a)  {
        Map<String,String> map = new HashMap<>();
                        map.put("Ravi", "12345");  //Ravi is key and 12345 is value
                        map.put("Rahul", "12345");
                        map.put("Aswin", "5678");
                        map.put(null, "6390");
                        map.put("Ravi","1529");

                        System.out.println(map);

        System.out.println(map.get(null));
                        System.out.println(map.get("Virat")); //null becoz key is not available
                        map.forEach((k, v) -> System.out.println("Key = " + k + ", Value = " + v));

    }}
```

Output is:

```
{Aswin=5678, null=6390, Rahul=12345, Ravi=1529}
6390
null
Key = Aswin, Value = 5678
Key = null, Value = 6390
Key = Rahul, Value = 12345
Key = Ravi, Value = 1529
```

[You can see the output that HashMap is not following the insertion order while storing the entries, hence HashMap is unordered.]

**Operations**: Inserting key and value pair, searching specific key and value, removing one entry, retrieving entry based on specific key, fetching data by using forEach method.

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

public class HashMapOperations {

   public static void main(String[] args) {
      // Creating a HashMap
      HashMap<String, Integer> hashMap = new HashMap<>();

      // Inserting key-value pairs
      hashMap.put("One", 1);
      hashMap.put("Two", 2);
      hashMap.put("Three", 3);
```

```
        hashMap.put("Four", 4);

        // Displaying the initial HashMap
        System.out.println("Initial HashMap: " + hashMap);

        // Searching for a specific key
        String searchKey = "Two";
        if (hashMap.containsKey(searchKey)) {
            System.out.println("Key '" + searchKey + "' found. Value: " + hashMap.get(searchKey));
        } else {
            System.out.println("Key '" + searchKey + "' not found.");
        }

        // Searching for a specific value
        int searchValue = 3;
        if (hashMap.containsValue(searchValue)) {
            System.out.println("Value '" + searchValue + "' found.");
        } else {
            System.out.println("Value '" + searchValue + "' not found.");
        }

        // Removing an entry
        String keyToRemove = "Three";
        if (hashMap.containsKey(keyToRemove)) {
            hashMap.remove(keyToRemove);
            System.out.println("Removed key '" + keyToRemove + "'. Updated HashMap: " + hashMap);
        } else {
            System.out.println("Key '" + keyToRemove + "' not found. No removal performed.");
        }

        // Retrieving an entry based on a specific key
        String specificKey = "One";
        int valueForSpecificKey = hashMap.getOrDefault(specificKey, -1);
        System.out.println("Value for key '" + specificKey + "': " + valueForSpecificKey);

        // Fetching data using forEach method
        System.out.println("Fetching data using forEach method:");
        hashMap.forEach((key, value) -> System.out.println(key + ": " + value));
    }
}
```

**Operations**: program to search specific keys and values.

```
import java.util.*;
public class HashMapDemo1{
        public static void main(String args[])
        {
                HashMap<Integer,String> hm = new HashMap<>();
                hm.put(1, "JSE");
                hm.put(2, "JEE");
                hm.put(3, "JME");
                hm.put(4,"JavaFX");
                hm.put(5,null);
```

```
                        hm.put(6,null);

                        System.out.println("Initial map elements: " + hm);
                        System.out.println("key 2 is present or not :"+hm.containsKey(2));

                        System.out.println("JME is present or not :"+hm.containsValue("JME"));

                        System.out.println("Size of Map : " + hm.size());
                        hm.clear();
                        System.out.println("Map elements after clear: " + hm);
            }
}
```

**Operations**: Write a program to return a set of keys, return a set of view of mapping and return a collection view of the values from a map.

```
import java.util.*;
public class HashMapDemo2
{
public static void main(String args[])
            {
                        Map<Integer,String> map = new HashMap<>();
                        map.put(1, "C");
                        map.put(2, "C++");
                        map.put(3, "Java");
                        map.put(4, ".net");

               //keySet() returns a set of keys
                  Set<Integer> keys = map.keySet();
              System.out.println(keys);

       //values() returns a collection view of the values from a map
         Collection <String> values = map.values();
         System.out.println(keys);

            map.forEach((k,v)->System.out.println("Key :"+k+" Value :"+v) );
        System.out.println("Return Old Object value :"+map.put(4,"Python"));

        // entrySet() returns a set of view of mapping                                  for(Map.Entry m :
map.entrySet())
                                    {
                                                System.out.println(m.getKey()+" $ "+m.getValue());
                                    }
            }
}
```

**Operations**: Write a program to merge two Map Collection.

```
import java.util.*;
public class HashMapDemo4
{
```

```
public static void main(String args[])
        {
                HashMap<Integer,String> newmap1 = new HashMap<>();
                HashMap<Integer,String> newmap2 = new HashMap<>();

                newmap1.put(1, "SCJP");
                newmap1.put(2, "is");
                newmap1.put(3, "best");

                System.out.println("Values in newmap1: "+ newmap1);
                newmap2.put(4, "Exam");
                newmap2.putAll(newmap1);// Copies all of the mappings from the newmap1 to this
map.
                newmap2.forEach((k,v)->System.out.println(k+" : "+v));
  }
}
```

**Operations**: Program to store Custom object in a Map as an entry.

```
//Employee.java

import java.util.*;
class Employee
{
        int eid;
        String ename;
        Employee(int eid, String ename)
        {
                this.eid = eid;
                this.ename = ename;
        }

  @Override
        public boolean equals(Object obj)  //obj = e2
        {
                if(obj instanceof Employee)
    {
                        Employee e2 = (Employee) obj;
                        if(this.eid == e2.eid && this.ename.equals(e2.ename))
                        {
                                return true;
                        }
                        else
                        {
                                return false;
                        }
          }
                else
                {

                        System.out.println("Comparison is not possible");
                }
        }
}


//HashMapDemo8.java
public class HashMapDemo8
{
        public static void main(String[] args)
        {
                Employee e1 = new Employee(101,"Aryan");
                Employee e2 = new Employee(102,"Pooja");
                Employee e3 = new Employee(101,"Aryan");
                Employee e4 = e2;

                HashMap<Employee,String> hm = new HashMap<>();
                hm.put(e1,"Ameerpet");
                hm.put(e2,"S.R Nagar");
                hm.put(e3,"Begumpet");
                hm.put(e4,"Panjagutta");

                hm.forEach((k,v)-> System.out.println(k+" : "+v));
        }
}
```

**Operations**: Program to adding null as an entry, removing specific key from a HashMap, insert an entry if the key is not present in a HashMap.

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

public class HashMapOperations {
  public static void main(String[] args) {
    // Create a HashMap
    Map<String, String> hashMap = new HashMap<>();

    // Adding null as an entry
    hashMap.put("Key1", "Value1");
    hashMap.put("Key2", null);
    hashMap.put("Key3", "Value3");

    // Display the HashMap
    System.out.println("HashMap after adding null:");
    displayHashMap(hashMap);

    // Removing a specific key
    String keyToRemove = "Key2";
    if (hashMap.containsKey(keyToRemove)) {
      hashMap.remove(keyToRemove);
      System.out.println("\nHashMap after removing key '" + keyToRemove + "':");
      displayHashMap(hashMap);
    } else {
      System.out.println("\nKey '" + keyToRemove + "' not found in the HashMap.");
    }

    // Inserting an entry if the key is not present using computeIfAbsent
```

```
        String keyToInsert = "Key4";
        String valueToInsert = "Value4";
        hashMap.computeIfAbsent(keyToInsert, k -> valueToInsert);

// Trying to insert a new entry for an existing key (no change)
                    hashmap.putIfAbsent("key1","value5");

        System.out.println("\nHashMap after inserting key '" + keyToInsert + "':");
        displayHashMap(hashMap);
    }

    // Display the content of the HashMap
    private static void displayHashMap(Map<String, String> hashMap) {
        for (Map.Entry<String, String> entry : hashMap.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }
    }
}
```

**LinkedHashMap:**

Public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>

- It is a predefined class available in java.util package under Map interface.
- It is the sub class of HashMap class.
- It maintains insertion order. It contains a doubly linked with the elements or nodes so it will iterate more slowly in comparison to HashMap.
- It uses Hashtable and LinkedList data structure.
- If we want to fetch the elements in the same order as they were inserted then we should go with LInkedHashMap.
- It accepts one null key and multiple null values.
- It is not synchronized.

**Constructors of LinkedHashMap:**

It has also 5 constructors same as HashMap

1. LinkedHashMap()
   Constructs an empty insertion-ordered LinkedHashMap instance with the default initial capacity (16) and load factor (0.75).
2. LinkedHashMap(int initialCapacity)
   Constructs an empty insertion-ordered LinkedHashMap instance with the specified initial capacity and a default load factor (0.75).
3. LinkedHashMap(int initialCapacity, float loadFactor)
   Constructs an empty insertion-ordered LinkedHashMap instance with the specified initial capacity and load factor.
4. LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)

Constructs an empty LinkedHashMap instance with the specified initial capacity, load factor and ordering mode.

5. LinkedHashMap(Map<? extends K,? extends V> m)
   Constructs an insertion-ordered LinkedHashMap instance with the same mappings as the specified map.

**Methods of LinkedHashMap:**

*Methods inherited from class java.util.HashMap :*

clone, compute, computeIfAbsent, computeIfPresent, containsKey, isEmpty, merge, put, putAll, putIf Absent, remove, remove, replace, replace, size

*Methods inherited from interface java.util.Map:*

compute, computeIfAbsent, computeIfPresent, containsKey, equals, hashCode, isEmpty, merge, put, putAll, putIfAbsent, remove, remove, replace, replace, size

*Overriding methods:*

a) public boolean containsValue(Object value): Returns true if this map maps one or more keys to the specified value.
b) public V get(Object key): Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
c) public V getOrDefault(Object key,V defaultValue): Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
d) public void clear(): Removes all of the mappings from this map. The map will be empty after this call returns.
e) protected boolean removeEldestEntry(Map.Entry<K,V> eldest): Returns true if this map should remove its eldest entry.
f) public Set<K> keySet(): Returns a Set view of the keys contained in this map.
g) public Collection<V> values(): Returns a Collection view of the values contained in this map.
h) public Set<Map.Entry<K,V>> entrySet(): Returns a Set view of the mappings contained in this map.
i) public void forEach(BiConsumer<? super K,? super V> action)

**Programs on LinkedHashMap:**
**Operations**: Program to create LinkedHashMap object using all constructors and adding key value pairs and displaying them.

```
import java.util.LinkedHashMap;
import java.util.Map;

public class LinkedHashMapExample {
   public static void main(String[] args) {
      // Create a LinkedHashMap using the default constructor
      Map<String, Integer> linkedHashMap1 = new LinkedHashMap<>();
      linkedHashMap1.put("One", 1);
      linkedHashMap1.put("Two", 2);
      linkedHashMap1.put("Three", 3);

      // Create a LinkedHashMap using the copy constructor
```

```
        Map<String, Integer> linkedHashMap2 = new LinkedHashMap<>(linkedHashMap1);
        linkedHashMap2.put("Four", 4);

        // Create a LinkedHashMap with an initial capacity and load factor
        int initialCapacity = 10;
        float loadFactor = 0.75f;
        Map<String, Integer> linkedHashMap3 = new LinkedHashMap<>(initialCapacity, loadFactor);
        linkedHashMap3.put("Five", 5);

        // Create a LinkedHashMap with an initial capacity, load factor, and order
        boolean accessOrder = true;
        Map<String, Integer> linkedHashMap4 = new LinkedHashMap<>(initialCapacity, loadFactor,
accessOrder);
        linkedHashMap4.put("Six", 6);

        // Display entries of each LinkedHashMap using forEach and lambda
        displayLinkedHashMap("LinkedHashMap 1:", linkedHashMap1);
        displayLinkedHashMap("LinkedHashMap 2:", linkedHashMap2);
        displayLinkedHashMap("LinkedHashMap 3:", linkedHashMap3);
        displayLinkedHashMap("LinkedHashMap 4:", linkedHashMap4);
    }

    // Display the content of the LinkedHashMap using forEach and lambda
    private static void displayLinkedHashMap(String message, Map<String, Integer> linkedHashMap) {
        System.out.println(message);
        linkedHashMap.forEach((key, value) -> System.out.println(key + ": " + value));
        System.out.println();
    }
}
```

**Operations**: Program to add, fetch, and compute entries on LinkedHashMap.

```
import java.util.LinkedHashMap;
import java.util.Map;

public class LinkedHashMapOperations {
    public static void main(String[] args) {
        // Create a LinkedHashMap
        Map<String, Double> productPrices = new LinkedHashMap<>();

        // Add entries to the LinkedHashMap
        addEntry(productPrices, "Laptop", 1200.0);
        addEntry(productPrices, "Smartphone", 800.0);
        addEntry(productPrices, "Tablet", 400.0);

        // Display the initial state of the LinkedHashMap
        displayLinkedHashMap("LinkedHashMap before compute:", productPrices);

        // Compute the updated price of a specific product
        String productNameToUpdate = "Smartphone";
        computeUpdatedPrice(productPrices, productNameToUpdate, (key, value) -> value * 0.9);
```

```
      // Display the LinkedHashMap after the compute operation
      displayLinkedHashMap("LinkedHashMap after compute:", productPrices);
   }

   // Add entry to the LinkedHashMap
   private static void addEntry(Map<String, Double> map, String key, double value) {
      map.put(key, value);
   }

   // Display the content of the LinkedHashMap
   private static void displayLinkedHashMap(String message, Map<String, Double> map) {
      System.out.println(message);
      map.forEach((key, value) -> System.out.println(key + ": $" + value));
      System.out.println();
   }

   // Compute the updated price of a specific product using the compute method
   private static void computeUpdatedPrice(Map<String, Double> map, String key, BiFunction<String,
Double, Double> function) {
      map.compute(key, (product, currentPrice) -> function.apply(product, currentPrice));
   }}
```

**Operations**: Program to use keyset, values, entrySet on LinkedhashMap.

```
import java.util.LinkedHashMap;
import java.util.Map;

public class LinkedHashMapIteration {
   public static void main(String[] args) {
      // Create a LinkedHashMap
      Map<String, Integer> studentScores = new LinkedHashMap<>();

      // Add entries to the LinkedHashMap
      addEntry(studentScores, "Alice", 85);
      addEntry(studentScores, "Bob", 92);
      addEntry(studentScores, "Charlie", 78);

      // Display the content of the LinkedHashMap using keySet
      displayKeySet("Keys in the LinkedHashMap:", studentScores);

      // Display the content of the LinkedHashMap using values
      displayValues("Values in the LinkedHashMap:", studentScores);

      // Display the content of the LinkedHashMap using entrySet
      displayEntrySet("Entries in the LinkedHashMap:", studentScores);
   }

   // Add entry to the LinkedHashMap
   private static void addEntry(Map<String, Integer> map, String key, int value) {
      map.put(key, value);
   }

   // Display keys in the LinkedHashMap using keySet
```

```
  private static void displayKeySet(String message, Map<String, Integer> map) {
    System.out.println(message);
    for (String key : map.keySet()) {
      System.out.println(key);
    }
    System.out.println();
  }

  // Display values in the LinkedHashMap using values
  private static void displayValues(String message, Map<String, Integer> map) {
    System.out.println(message);
    for (int value : map.values()) {
      System.out.println(value);
    }
    System.out.println();
  }

  // Display entries in the LinkedHashMap using entrySet
  private static void displayEntrySet(String message, Map<String, Integer> map) {
    System.out.println(message);
    for (Map.Entry<String, Integer> entry : map.entrySet()) {
      System.out.println(entry.getKey() + ": " + entry.getValue());
    }
    System.out.println();
  }
}
```

**Hashtable:**

public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>, Cloneable, Serializable

- It is predefined class available in java.util package under Map interface.
- Like Vector, Hashtable is also form the birth of java so called legacy class.
- It is the sub class of Dictionary class which is an abstract class.
- This class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value.
- The objects used as keys must implement the hashCode method and the equals method.

**Constructors of Hashtable:**

1) Hashtable hs1 = new Hashtable() : It will create the Hashtable Object with default capacity as 11 as well as load factor is 0.75

2) Hashtable hs2 = new Hashtable(int initialCapacity) : will create the Hashtable object with specified capacity

3) Hashtable hs3 = new Hashtable(int initialCapacity, float loadFactor) : we can specify our own initialCapacity and loadFactor

4) Hashtable hs = new Hashtable(Map c) : Interconversion of Map Collection.

**Programs on Hashtable:**

**Operations:**

Program to create hashtable using all its constructors.

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

public class HashtableOperations {
  public static void main(String[] args) {
    // Creating a Hashtable using the default constructor
    Map<String, Integer> numbers = new Hashtable<>();

    // Adding entries to the Hashtable
    numbers.put("One", 1);
    numbers.put("Two", 2);
    numbers.put("Three", 3);

    // Displaying the content of the Hashtable using Iterator
    System.out.println("Display Hashtable using Iterator:");
    Iterator<Map.Entry<String, Integer>> iterator = numbers.entrySet().iterator();
    while (iterator.hasNext()) {
      Map.Entry<String, Integer> entry = iterator.next();
      System.out.println(entry.getKey() + ": " + entry.getValue());
    }

   // Creating a Hashtable using a single-parameter constructor
    Hashtable<String, String> colors = new Hashtable<>(10);

    // Adding entries to the Hashtable
    colors.put("Red", "#FF0000");
    colors.put("Green", "#00FF00");
    colors.put("Blue", "#0000FF");

    // Displaying the content of the Hashtable using Enumeration
    System.out.println("Display Hashtable using Enumeration:");
    Enumeration<String> keys = colors.keys();
    while (keys.hasMoreElements()) {
      String key = keys.nextElement();
      System.out.println(key + ": " + colors.get(key));
    }

   // Creating a Hashtable using a two-parameter constructor
    Map<String, Integer> ages = new Hashtable<>(10, 0.75f);

    // Adding entries to the Hashtable
    ages.put("John", 25);
    ages.put("Jane", 30);
    ages.put("Bob", 22);

    // Displaying the content of the Hashtable using forEach with lambda
    System.out.println("Display Hashtable using forEach with lambda:");
```

```
        ages.forEach((name, age) -> System.out.println(name + ": " + age));

       // Creating a HashMap
        Map<String, String> countryCodes = new HashMap<>();
        countryCodes.put("USA", "+1");
        countryCodes.put("UK", "+44");
        countryCodes.put("India", "+91");

        // Creating a Hashtable using the constructor that takes a Map parameter
        Hashtable<String, String> countryCodeTable = new Hashtable<>(countryCodes);

        // Displaying the content of the Hashtable using forEach with method reference
        System.out.println("Display Hashtable using forEach with method reference:");
        countryCodeTable.forEach(System.out::println);
    }
}
```

**Operations**: Program to use entryset and Entry(I)

```
import java.util.*;
public class HashtableDemo{
        public static void main(String args[]){
                Hashtable<Integer,String> map=new Hashtable<>();
                map.put(1, "Java");
                map.put(2, "is");
                map.put(3, "best");
                map.put(4,"language");

                //map.put(5,null);
                System.out.println(map);
                System.out.println(" ....................");
                for(Map.Entry m : map.entrySet())
                        {
                                System.out.println(m.getKey()+" = "+m.getValue());
                        }
    }
}
```

***[Note: Map is an interface and Entry is also an interface defined inside Map interface to create an entry]***

```
interface Map {
        interface Entry{
                //key and value pair
        }
}
```

**Operations**: Program to add some key if that key is not present.

```
import java.util.*;
public class HashtableDemo1
{
  public static void main(String args[])
```

```
          {
   Hashtable<Integer,String> map=new Hashtable<>();
    map.put(1,"Priyanka");
    map.put(2,"Ruby");
    map.put(3,"Vibha");
    map.put(4,"Kanchan");

           map.putIfAbsent(5,"Bina");
           map.putIfAbsent(24,"Pooja");
           map.putIfAbsent(26,"Ankita");

    map.putIfAbsent(1,"Sneha");
    System.out.println("Updated Map: "+map);
 }
}
```

Difference between HashMap and Hashtable.

| HashMap | Hashtable |
|---|---|
| HashMap is introduced in version 1.2.<br>So it is Non-legacy. | Hashtable is introduced in version 1.0. So it is considered as legacy. |
| HashMap is not synchronized, therefor not particularly thread-safe. | Hashtable is synchronized, therefore be used in multithreaded implementations. |
| It can have one null key and many null values. | Does not permit null keys or values. Inserting null can lead to a NullPointerException. |
| It is faster than the Hashtable and uses less memory. | It is much slower than the HashMap |
| The iterator is fail-fast.<br>Throws ConcurrentModificationException if modified by another thread during iteration. | Enumerator is not fail-fast. Due to internal synchronization, concurrent modification risks are minimized during enumeration. |
| Part of the collection framework from its introduction. Implements the **Map** interface. | Initially not part of the collection framework. Later integrated after implementing the **Map** interface. |
| Can be used to implement LinkedHashMap (maintains insertion order) and TreeMap (sorted order). | Does not guarantee any specific order for its entries. |

**Properties:**

Properties class is used to maintain the data in the key-value form. It takes both key and value as a string. Properties class is a subclass of Hashtable. It provides the methods to store properties in a properties file and to get the properties from the properties file. System.getProperties() returns the all system’s properties.

**Constructors of Properties:**

1. **Properties**(): Creates an empty property list with no default values.

2. **Properties**(**Properties** defaults): Creates an empty property list with the specified defaults.

**Methods of Properties:**

a) public Object setProperty(String key, String value): Calls the Hashtable method put. Provided for parallelism with the getProperty method. Enforces use of strings for property keys and values. The value returned is the result of the Hashtable call to put.
b) public void load(Reader reader) throws IOException: Reads a property list (key and element pairs) from the input character stream in a simple line-oriented format.
c) public void load(InputStream inStream) throws IOException: Reads a property list (key and element pairs) from the input byte stream.
d) public void store(Writer writer, String comments)throws IOException: Writes this property list (key and element pairs) in this Properties table to the output character stream in a format suitable for using the load(Reader) method.
e) public void store(OutputStream out, String comments) throws IOException: Writes this property list (key and element pairs) in this Properties table to the output stream in a format suitable for loading into a Properties table using the load(InputStream) method.
f) public String getProperty(String key): Searches for the property with the specified key in this property list
g) public String getProperty(String key, String defaultValue): Searches for the property with the specified key in this property list.
h) public Enumeration<?> propertyNames(): Returns an enumeration of all the keys in this property list, including distinct keys in the default property list if a key of the same name has not already been found from the main properties list.
i) public Set<String> stringPropertyNames()
j) public void list(PrintStream out): Prints this property list out to the specified output stream. This method is useful for debugging.
k) public void list(PrintWriter out): Prints this property list out to the specified output stream. This method is useful for debugging.

**Programs on Properties class:**

**Operations**: Program to use Properties class.

```
db.properties
---------
 driver = oracle.jdbc.driver.OracleDriver
 user = system
 password = tiger
------------------------------------------
import java.util.*;
import java.io.*;
public class PropertiesExample1
{
public static void main(String[] args)throws Exception
 {
   FileReader reader=new FileReader("db.properties");

   Properties p=new Properties();
   p.load(reader);

   System.out.println(p.getProperty("user"));
   System.out.println(p.getProperty("password"));
         System.out.println(p.getProperty("driver"));
 }
}
```

**Operations**: Program to show the use of Map.entry and getValue.

```
import java.util.*;
import java.io.*;
public class PropertiesExample2
{
public static void main(String[] args)throws Exception
{
        Properties p=System.getProperties();
        Set set=p.entrySet();
        Iterator itr=set.iterator();
        while(itr.hasNext())
  {
                Map.Entry entry=(Map.Entry)itr.next();
                System.out.println(entry.getKey()+" = "+entry.getValue());
        }
 }
```

**IdentityHashMap:**

public class IdentityHashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Clonable, Serializable.

- It was introduced from JDK 1.4 onwards.
- The IdentityHashMap uses == operator to compare keys.
- As we know HashMap uses equals() and hashCode() method for comparing the keys based on the hashcode of the object it will search the bucket location and insert the entry their only.
- So We should use IdentityHashMap where we need to check the reference or memory address instead of logical equality.
- HashMap uses hashCode of the "Object key" to find out the bucket loaction in Hashtable, on the other hand IdentityHashMap does not use hashCode() method actually It uses System.identityHashCode(Object o)
- IdentityHashMap is more faster than HashMap in case of Comparison.
- Allows only single null key.
- It has three constrcutors, It does not contain loadFactor specific constructor.

**Constructors of identityHashMap:**

1. IdentityHashMap(): Constructs a new, empty identity hash map with a default expected maximum size (21).
2. IdentityHashMap(int expectedMaxSize): Constructs a new, empty map with the specified expected maximum size.
3. IdentityHashMap(Map<? extends K,? extends V> m):
   Constructs a new identity hash map containing the keys-value mappings in the specified map.

**Programs on IdentityHashMap:**

**Operation**: Program to check that the IdenticalHashMap stores the object by checking their address but not by contents.

```
import java.util.*;
public class IdentityHashMapDemo
{
        public static void main(String[] args)
        {
                HashMap<String,Integer> hm = new HashMap<>();


        IdentityHashMap<String,Integer> ihm = new IdentityHashMap<>();

                hm.put("Ravi",23);
                hm.put(new String("Ravi"), 24);

                ihm.put("Ravi",23);
                ihm.put(new String("Ravi"), 27); //compares based on == operator

                System.out.println("HashMap size :"+hm.size());  //1
                System.out.println(hm);
                System.out.println(" ..................... ");
                System.out.println("IdentityHashMap size :"+ihm.size()); //2
                System.out.println(ihm);
        }
}
```

**Operations**: Program to store data and fetch data.

```
import java.util.IdentityHashMap;
import java.util.Map;

public class IdentityHashMapExample {
    public static void main(String[] args) {
        // Creating an IdentityHashMap
        Map<String, String> identityMap = new IdentityHashMap<>();

        // Adding key-value pairs
        String key1 = new String("John");
        String key2 = new String("John");

        identityMap.put(key1, "Value1");
        identityMap.put(key2, "Value2");

        // Size of the map (1, not 2, because key1 and key2 are different objects)
        System.out.println("Size of IdentityHashMap: " + identityMap.size());

        // Accessing values using keys
        System.out.println("Value for key1: " + identityMap.get(key1));
        System.out.println("Value for key2: " + identityMap.get(key2));
        // Iterating over entries using forEach and lambda expression
        System.out.println("Entries in IdentityHashMap:");
        identityMap.forEach((k, v) -> System.out.println("Key: " + k + ", Value: " + v));
    }
}
```

**WeakHashMap**:

public class WeakHashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>

- It is a predefined class in java.util package under Map interface.It was introduced from JDK 1.2v onwards.
- While working with HashMap, keys of HashMap are of strong reference type. This means the entry of map will not be deleted by the garbage collector even though the key is set to be null and still it is not eligible for Garbage Collector.
- On the other hand while working with WeakHashMap, keys of WeakHashMap are of weak reference type. This means the entry of a map is deleted by the garbage collector if the key value is set to be null because it is of weak type.
- So, HashMap dominates over Garbage Collector where as Garbage Collector dominates over WeakHashMap.
- It contains 4 types of Constructor.

**Constructor of WeakHashMap:**

1. WeakHashMap()
   Constructs a new, empty WeakHashMap with the default initial capacity (16) and load factor (0.75).
2. WeakHashMap(int initialCapacity)
   Constructs a new, empty WeakHashMap with the given initial capacity and the default load factor (0.75).
3. WeakHashMap(int initialCapacity, float loadFactor)
   Constructs a new, empty WeakHashMap with the given initial capacity and the given load factor.
4. WeakHashMap(Map<? extends K,? extends V> m)
   Constructs a ne WeakHashMap with the same mappings as the specified map.

*LoadFactor* - The load factor of this map is 0.9. This means whenever our hashtable is filled up by 90%, the entries are moved to a new hashtable of double the size of the original hashtable.

**Programs on WeakHashMap**:

**Operations**: Program to show the use of WeakHashMap.

```
import java.util.*;
public class WeakHashMapDemo
{
        public static void main(String args[])  throws Exception
        {
          WeakHashMap<Test,String> map = new WeakHashMap<>();

                Test  t  = new Test();
                map.put(t," Rahul "); //here we passing reference 't' as a key
                System.out.println(map); //{Test Nit = Rahul}

                t = null;

                System.gc();      //Explicitly calling garbage collector

                Thread.sleep(5000);

                System.out.println(map); //{}
        }
}

class Test {
        @Override
        public String toString() {
                return "Test Nit";
        }

        @Override
        public void finalize() //called automatically if an object is eligible 4 GC
        {
                System.out.println("finalize method is called");
        }
}
```

*Explanation of the Program:*

An instance of WeakHashMap is created, and a Test object (t) is used as a key.

The WeakHashMap is printed, showing the mapping {Test Nit = Rahul}.

The reference to t is set to null explicitly, making it eligible for garbage collection.

The garbage collector is explicitly called using System.gc().

The program sleeps for 5 seconds to allow the garbage collector to act.

After 5 seconds, the WeakHashMap is printed again, and since the key t is now eligible for garbage collection, the map becomes empty ({}). Additionally, the finalize method of the Test class is called when the object is garbage collected.

This example demonstrates the behavior of WeakHashMap with weak references and how entries are automatically removed when keys become unreachable.

**SortedMap(I):**

- public interface SortedMap<K,V> extends Map<K,V>
- It is a predefined interface available in java.util package under Map interface.
- We should use SortedMap interface when we want to insert the key element based on some sorting order i.e the default natural sorting order.
- All general-purpose sorted map implementation classes should provide four "standard" constructors.
- The expected "standard" constructors for all sorted map implementations are:

1. A void (no arguments) constructor, which creates an empty sorted map sorted according to the natural ordering of its keys.
2. A constructor with a single argument of type Comparator, which creates an empty sorted map sorted according to the specified comparator.
3. A constructor with a single argument of type Map, which creates a new map with the same key-value mappings as its argument, sorted according to the keys' natural ordering.
4. A constructor with a single argument of type SortedMap, which creates a new sorted map with the same key-value mappings and the same ordering as the input sorted map.

**Methods of SortedMap:**

a) Comparator<? super K> comparator(): Returns the comparator used to order the keys in this map, or null if this map uses the natural ordering of its keys.
b) SortedMap<K,V> subMap(K fromKey, K toKey): Returns a view of the portion of this map whose keys range from fromKey.
c) SortedMap<K,V> headMap(K toKey): Returns a view of the portion of this map whose keys are strictly less than toKey.
d) SortedMap<K,V> tailMap(K fromKey): Returns a view of the portion of this map whose keys are greater than or equal to from Key.
e) K firstKey(): Returns the first (lowest) key currently in this map.
f) K lastKey(): Returns the last (highest) key currently in this map.
g) Set<K> keySet(): Returns a Set view of the keys contained in this map.
h) Collection<V> values(): Returns a Collection view of the values contained in this map.
i) Set<Map.Entry<K,V>> entrySet(): Returns a Set view of the mappings contained in this map.

**TreeMap:**

public class TreeMap<K,V> extends AbstractMap<K,V> implements NavigableMap<K,V> , Clonable, Serializable

- It is a predefined class avaialble in java.util package under Map interface.
- It is a sorted map that means it will sort the elements by natural sorting order based on the key using Comparator interface.
- It will accept homogeneous keys (comparable keys as an Object) only because it will sort the entry of TreeMap based on the key.
- It does not accept null key but null value allowed.
- TreeMap implements NavigableMap and NavigableMap extends SortedMap. SortedMap extends Map interface.
- TreeMap contains 4 types of Constructors .

**Constructors of TreeMap:**

1. TreeMap()
   Constructs a new, empty tree map, using the natural ordering of its keys.
2. TreeMap(Comparator<? super K> comparator)
   Constructs a new, empty tree map, ordered according to the given comparator.
3. TreeMap(Map<? extends K,? extends V> m)
   Constructs a new tree map containing the same mappings as the given map, ordered according to the *natural ordering* of its keys.
4. TreeMap(SortedMap<K,? extends V> m)

Constructs a new tree map containing the same mappings and using the same ordering as the specified sorted map.

**Methods of TreeMap:** All methods are inherited from AbstractMap,Map(I) ,SortedMap(I) and NavigableMap.

a) public Map.Entry<K,V> firstEntry(): Returns a key-value mapping associated with the least key in this map, or null if the map is empty.
b) public Map.Entry<K,V> lastEntry(): Returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
c) public Map.Entry<K,V> pollFirstEntry(): Removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty.
d) public Map.Entry<K,V> pollLastEntry(): Removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
e) public Map.Entry<K,V> lowerEntry(K key): Returns a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key.
f) public K lowerKey(K key): Returns the greatest key strictly less than the given key, or null if there is no such key.
g) public Map.Entry<K,V> floorEntry(K key): Returns a key-value mapping associated with the greatest key less than or equal to the given key, or null if there is no such key.
h) public K floorKey(K key): Returns the greatest key less than or equal to the given key, or null if there is no such key.
i) public Map.Entry<K,V> ceilingEntry(K key): Returns a key-value mapping associated with the least key greater than or equal to the given key, or null if there is no such key.
j) public K ceilingKey(K key): Returns the least key greater than or equal to the given key, or null if there is no such key.
k) public K higherKey(K key): Returns the least key strictly greater than the given key, or null if there is no such key.
l) public SortedMap<K,V> subMap(K fromKey, K toKey): Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive.
m) public SortedMap<K,V> headMap(K toKey): Returns a view of the portion of this map whose keys are strictly less than toKey.
n) public SortedMap<K,V> tailMap(K fromKey): Returns a view of the portion of this map whose keys are greater than or equal to fromKey.

Etc…………

**Programs on TreeMap:**
**Operations**: Program to show the use of all Constructors of TreeMap.

```
import java.util.Comparator;
import java.util.TreeMap;

public class TreeMapExample {

  public static void main(String[] args) {
    // Default Constructor
    TreeMap<Integer, String> treeMap1 = new TreeMap<>();
    treeMap1.put(3, "Three");
    treeMap1.put(1, "One");
    treeMap1.put(2, "Two");
    System.out.println("TreeMap 1: " + treeMap1);

    // Constructor with Comparator
```

```
        Comparator<Integer> customComparator = (a, b) -> b.compareTo(a); // Reverse order
        TreeMap<Integer, String> treeMap2 = new TreeMap<>(customComparator);
        treeMap2.put(3, "Three");
        treeMap2.put(1, "One");
        treeMap2.put(2, "Two");
        System.out.println("TreeMap 2 (Custom Comparator): " + treeMap2);

        // Constructor with Map
        TreeMap<Integer, String> treeMap3 = new TreeMap<>(treeMap1);
        System.out.println("TreeMap 3 (Copy of TreeMap 1): " + treeMap3);

        // Constructor with Comparator and Map
        TreeMap<Integer, String> treeMap4 = new TreeMap<>(customComparator);
        treeMap4.putAll(treeMap1);
        System.out.println("TreeMap 4 (Custom Comparator and Copy of TreeMap 1): " + treeMap4);
```

**Operation**: Program to store all the keys in a natural sorting order.

```
import java.util.*;
public class TreeMapDemo
{
        public static void main(String[] args)
        {
                TreeMap t = new TreeMap();
                t.put(4,"Ravi");
                t.put(7,"Aswin");
                t.put(2,"Ananya");
                t.put(1,"Dinesh");
                t.put(9,"Ravi");
                t.put(3,"Ankita");
                t.put(5,null);
                System.out.println(t);
        }
}
```

**Operation**: Program to store key and value and fetch it by using forEachremaining and forEach.

```
import java.util.*;
public class TreeMapDemo1
{
    public static void main(String args[])
    {
        TreeMap map = new TreeMap();
        map.put("one","1");
        map.put("two",null);
        map.put("three","3");
        map.put("four",4);
        displayMap(map);
        map.forEach((k, v) -> System.out.println("Key = " + k + ", Value = " + v));
    }
    static void displayMap(TreeMap map)
    {
       Collection c = map.entrySet();  //Set<Map.Entry>
```

```
        Iterator i = c.iterator();
        i.forEachRemaining(x -> System.out.println(x));
    }
}
```

**Operations**: Program to see the first key and last key of a TreeMap and show the use ofheadMap(), submap(), tailMap().

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

public class TreeMapOperations {

    public static void main(String[] args) {
        // Creating a TreeMap
        TreeMap<Integer, String> treeMap = new TreeMap<>();

        // Adding key-value pairs to the TreeMap
        treeMap.put(1, "One");
        treeMap.put(2, "Two");
        treeMap.put(3, "Three");
        treeMap.put(4, "Four");
        treeMap.put(5, "Five");

        // Displaying the original TreeMap
        System.out.println("Original TreeMap: " + treeMap);

        // Retrieving the first key
        int firstKey = treeMap.firstKey();
        System.out.println("First Key: " + firstKey);

        // Retrieving the last key
        int lastKey = treeMap.lastKey();
        System.out.println("Last Key: " + lastKey);

        // Retrieving the head of the map (entries less than a specified key)
        Map<Integer, String> headMap = treeMap.headMap(3);
        System.out.println("Head Map (entries less than key 3): " + headMap);

        // Retrieving the tail of the map (entries greater than or equal to a specified key)
        Map<Integer, String> tailMap = treeMap.tailMap(3);
        System.out.println("Tail Map (entries greater than or equal to key 3): " + tailMap);

        // Retrieving a submap (entries between two specified keys)
        Map<Integer, String> subMap = treeMap.subMap(2, 4);
        System.out.println("SubMap (entries between keys 2 (inclusive) and 4 (exclusive)): " + subMap);
    }
}
```

**Operations**: Working with Custom Object in TreeMap.

```
import java.util.TreeMap;
```

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

		System.out.println("\t***********Sorting name -> Ascending
Order***********\t");

		TreeMap<Employee,String> tm1 = new TreeMap<>((e1,e2)-
>e1.name.compareTo(e2.name));

		tm1.put(new Employee(101, "Zaheer", 24),"Hyderabad");
		tm1.put(new Employee(201, "Aryan", 27),"Jamshedpur");
		tm1.put(new Employee(301, "Pooja", 26),"Mumbai");

		System.out.println(tm1);

		System.out.println();

		System.out.println("\t***********Sorting name -> Descending
Order***********\t");

		TreeMap<Employee,String> tm2 = new TreeMap<>((e1,e2)->-
(e1.name).compareTo(e2.name));

		tm2.put(new Employee(101, "Zaheer", 24),"Hyderabad");
		tm2.put(new Employee(201, "Aryan", 27),"Jamshedpur");
		tm2.put(new Employee(301, "Pooja", 26),"Mumbai");

		System.out.println(tm2);

		System.out.println();
		System.out.println("\t***********Sorting Age -> Ascending Order***********\t");

		TreeMap<Employee,String> tm3 = new TreeMap<>((e1,e2)-
>e1.age.compareTo(e2.age));
		tm3.put(new Employee(101, "Zaheer", 24),"Hyderabad");
		tm3.put(new Employee(201, "Aryan", 27),"Jamshedpur");
		tm3.put(new Employee(301, "Pooja", 26),"Mumbai");

		System.out.println(tm3);

		System.out.println();
		System.out.println("\t***********Sorting Age ->  Descending
Order***********\t");
		TreeMap<Employee,String> tm4 = new TreeMap<>((e1,e2)->-
(e1.age).compareTo(e2.age));

		tm4.put(new Employee(101, "Zaheer", 24),"Hyderabad");
		tm4.put(new Employee(201, "Aryan", 27),"Jamshedpur");
		tm4.put(new Employee(301, "Pooja", 26),"Mumbai");
```

```
                    System.out.println(tm4);
        }
}

class Employee
        {
         int id;
         String name;
         Integer age;

        public Employee(int id, String name, int age)
        {
                this.id = id;
                this.name = name;
                this.age = age;
        }

        @Override
        public String toString()
        {
                return " " + this.id + " " + this.name + " "+ this.age;
        }
}
```

**NavigableMap interface:**

public interface NavigableMap<K,V> extends SortedMap<K,V>
NavigableMap is a sub-interface of the SortedMap interface in Java. It extends SortedMap to provide navigation methods for accessing the elements based on their order. NavigableMap is available in the java.util package. Here are some key features and methods of NavigableMap.

**Navigation Methods:**

a) lowerKey(K key): Returns the greatest key strictly less than the given key, or null if there is no such key.
b) floorKey(K key): Returns the greatest key less than or equal to the given key, or null if there is no such key.
c) ceilingKey(K key): Returns the least key greater than or equal to the given key, or null if there is no such key.
d) higherKey(K key): Returns the least key strictly greater than the given key, or null if there is no such key.
e) descendingKeySet(): Returns a reverse order NavigableSet view of the keys.

**Submap Views:**

a) subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive): Returns a view of the portion of this map whose keys range from fromKey to toKey.
b) headMap(K toKey, boolean inclusive): Returns a view of the portion of this map whose keys are strictly less than toKey.
c) tailMap(K fromKey, boolean inclusive): Returns a view of the portion of this map whose keys are greater than or equal to fromKey.

**Programs on NavigableMap:**

**Operation**: Demonstrate the usage of various methods provided by the NavigableMap.

```
import java.util.*;

public class NavigableMapExample {
   public static void main(String[] args) {
      NavigableMap<Integer, String> navigableMap = new TreeMap<>();

      navigableMap.put(3, "Three");
      navigableMap.put(1, "One");
      navigableMap.put(5, "Five");
      navigableMap.put(2, "Two");
      navigableMap.put(4, "Four");

      System.out.println("Original NavigableMap: " + navigableMap);

      // Navigation methods
      System.out.println("lowerKey(3): " + navigableMap.lowerKey(3));
      System.out.println("floorKey(3): " + navigableMap.floorKey(3));
      System.out.println("ceilingKey(3): " + navigableMap.ceilingKey(3));
      System.out.println("higherKey(3): " + navigableMap.higherKey(3));

      // Submap views
      System.out.println("subMap(2, true, 4, true): " + navigableMap.subMap(2, true, 4, true));
      System.out.println("headMap(3, true): " + navigableMap.headMap(3, true));
      System.out.println("tailMap(3, true): " + navigableMap.tailMap(3, true));

      // Descending key set
      NavigableSet<Integer> descendingKeySet = navigableMap.descendingKeySet();
      System.out.println("Descending Key Set: " + descendingKeySet);
   }
}
```

**Queue interface:**

public interface Queue<E>extends Collection<E>

1) It is sub interface of Collection(I)
2) It works in FIFO(First In first out)
3) It is an ordered collection.
4) In a queue, insertion is possible from last is called REAR where as deletion is possible from the starting is called FRONT of the queue.
5) From jdk 1.5 onwards LinkedList class implements Queue interface to handle the basic queue operations.
6) Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations.

Summary of Queue methods

| | *Throws exception* | *Returns special value* |
|---|---|---|
| **Insert** | add(e) | offer(e) |
| **Remove** | remove() | poll() |
| **Examine** | element() | peek() |

7) Queue implementations generally do not allow insertion of null elements, although some implementations, such as LinkedList.

**Methods of Queue(I):**

a) boolean add(E e)
Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available.

b) boolean offer(E e)
Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.

c) E remove()
Retrieves and removes the head of this queue.

d) E poll()
Retrieves and removes the head of this queue, or returns null if this queue is empty.

e) E element()
Retrieves, but does not remove, the head of this queue.

f) E peek()
Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

**Programs on Queue(I):**

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

public class QueueExample {
   public static void main(String[] args) {
      Queue<String> queue = new LinkedList<>();

      // Adding elements to the queue
      queue.offer("One");
      queue.offer("Two");
      queue.offer("Three");

      System.out.println("Original Queue: " + queue);

      // Removing elements from the queue
      String removedElement = queue.poll();
      System.out.println("Removed Element: " + removedElement);
      System.out.println("Queue after poll: " + queue);
```

```
        // Peeking at the head of the queue
        String peekedElement = queue.peek();
        System.out.println("Peeked Element: " + peekedElement);
        System.out.println("Queue after peek: " + queue);
    }
}
```

**PriorityQueue:**

public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable

1. It is a predefined class in java.util package, available from Jdk 1.5 onwards.
2. The LinkedList class has been enhanced to implement Queue interface to handle basic operation of Queue.
3. A PriorityQueue is used when we want to store the Objects based on some priority.
4. The main purpose of PriorityQueue class is to create a "Priority in, Priority out" queue which is opposite to classical FIFO order.
5. A priority Queue stores the element in a natural sorting order where the elements which are sorted first, will be accessed first from top the queue.
6. A priority queue does not permit null elements as well as It uses Comparator interface to sort the elements.
7. It provides natural sorting order so we can't take non-comparable objects(hetrogeneous types of Object)
8. The initial capacity of PriorityQueue is 11.

**Constructors of Priority Queue:**

1. PriorityQueue(): Creates a PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering.
2. PriorityQueue(Collection<? extends E> c): Creates a PriorityQueue containing the elements in the specified collection.
3. PriorityQueue(Comparator<? super E> comparator): Creates a PriorityQueue with the default initial capacity and whose elements are ordered according to the specified comparator.
4. PriorityQueue(int initialCapacity): Creates a PriorityQueue with the specified initial capacity that orders its elements according to their natural ordering.
5. PriorityQueue(int initialCapacity, Comparator<? super E> comparator): Creates a PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.
6. PriorityQueue(PriorityQueue<? extends E> c): Creates a PriorityQueue containing the elements in the specified priority queue.
7. PriorityQueue(SortedSet<? extends E> c): Creates a PriorityQueue containing the elements in the specified sorted set.

**Methods of PriorityQueue:**

a) public boolean add(E e): Inserts the specified element into this priority queue.
b) public boolean offer(E e):Inserts the specified element into this priority queue.
c) public E peek(): Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
d) public boolean remove(Object o): Removes a single instance of the specified element from this queue, if it is present.
e) public boolean contains(Object o)Returns true if this queue contains the specified element.

f) public Object[] toArray(): Returns an array containing all of the elements in this queue.

g) public <T> T[] toArray(T[] a): Returns an array containing all of the elements in this queue.

h) public int size(): Returns the number of elements in this collection.

i) public void clear(): Removes all of the elements from this priority queue. The queue will be empty after this call returns.

j) public E poll(): Retrieves and removes the head of this queue, or returns null if this queue is empty.

k) public final Spliterator<E> spliterator(): Creates a *late-binding* and *fail-fast* Spliterator over the elements in this queue.

Etc…..

**Programs on PriorityQueue:**

**Operation**: Program to show the usage of poll, peek methods in PriorityQueue.

```
public class PriorityQueueDemo
{
    public static void main(String[] argv)
    {
        PriorityQueue<String> pq = new PriorityQueue<>();
        pq.add("Orange");
                                    pq.add("Apple");
                                    pq.add("Mango");
                                    pq.add("Guava");

                                    System.out.println(pq.peek()); //Apple
                                    System.out.println(pq.poll()); //Apple
                                    System.out.println(pq.peek()); //Orange
                                    System.out.println(pq);          //[Orange]

    }
}
```

**[Note:- It stores the elements on basis of Priority in and priority out which internally maintains priority heap tree order.]**

**Operation**: Program to show the usage of remove, offer in PriorityQueue.

```
import java.util.PriorityQueue;
public class PriorityQueueDemo1
{
    public static void main(String[] argv)
    {
        PriorityQueue<String> pq = new PriorityQueue<>();
        pq.add("9");
        pq.add("8");
                                    pq.add("7");
        System.out.print(pq.peek() + " ");
        pq.offer("6");
                                    pq.offer("5");
```

```
            pq.add("3");

            pq.remove("1");
            System.out.print(pq.poll() + " ");
            if (pq.remove("2"))
               System.out.print(pq.poll() + " ");
            System.out.println(pq.poll() + " " + pq.peek());
                                System.out.println(pq);
        }
    }
```

**Operation**: using add method to add some Integer.

```
import java.util.PriorityQueue;

public class PriorityQueueDemo3 {
   public static void main(String[] argv) {
      PriorityQueue<Integer> pq = new PriorityQueue<>();
      pq.add(11);
      pq.add(2);
      pq.add(4);
      pq.add(6);
      System.out.println(pq);
   }
}
```

**Operation**: Using peek(),poll(),remove()

```
import java.util.PriorityQueue;

public class PriorityQueueDemo2 {
   public static void main(String[] argv) {
      PriorityQueue<String> pq = new PriorityQueue<>();
      pq.add("2");
      pq.add("4");
      pq.add("6");
      System.out.print(pq.peek() + " ");  // 2
      pq.offer("1");
      pq.offer("9");
      pq.add("3");

      pq.remove("1");
      System.out.print(pq.poll() + " ");  // 2
      if (pq.remove("2"))
         System.out.print(pq.poll() + " ");  // 3
      System.out.println(pq.poll() + " " + pq.peek() + " " + pq.poll());  // 4 6 9
   }
}
```

**Generics:**

Why generic came into picture :

As we know our compiler is known for Strict type checking because java is a statically typed checked language.

The basic problem with collection is It can hold any kind of Object.

```
ArrayList al = new ArrayList();
al.add("Ravi");
al.add("Aswin");
al.add("Rahul");
al.add("Raj");
al.add("Samir");

for(int i =0; i<al.size(); i++)
{
  String s = (String) al.get(i);
  System.out.println(s);
}
```

By looking the above code it is clear that Collection stores everything in the form of Object so here even after adding String type only we need type casting as shown below.

---

```
import java.util.*;
class Test1
{
	public static void main(String[] args)
	{
		ArrayList al = new ArrayList(); //raw type
		al.add("Ravi");
		al.add("Ajay");
		al.add("Vijay");

		for(int i=0; i<al.size(); i++)
		{
		String name = (String) al.get(i); //type casting is required
		System.out.println(name.toUpperCase());
		}
	}
}
```

Even after type casting there is no guarantee that the things which are coming from ArrayList Object is String only because we can add anything in the Collection as a result java.lang.ClassCastException as shown in the program below.

---

```
 import java.util.*;
class Test2
{
	public static void main(String[] args)
	{
		ArrayList t = new ArrayList(); //raw type
		t.add("alpha");
		t.add("beta");
		for (int i = 0; i < t.size(); i++)
```

```
                    {
                     String str =(String) t.get(i);
                     System.out.println(str);
                    }

                    t.add(1234);
                    t.add(1256);
                     for (int i = 0; i < t.size(); ++i)
                {
                             String obj= (String)t.get(i); //we can't perform type casting here
                             System.out.println(obj);
                     }
           }
}
```

-----------------------------------------------------------------------

To avoid all the above said problem Generics came into picture from JDK 1.5 onwards

-> It deals with type safe Object so there is a gurantee of both the end i.e putting inside and getting out.

Example:-
ArrayList<String > al = new ArrayList<>();

Now here we have a gurantee that only String can be inserted as well as only String will come out from the Collection so we can perform String related operation.

**Advantages:**-

a) Type safe Object (No compiler warning)

b) Strict compile time checking (Type erasure)

c) No need of type casting

-----------------------------------------------------------------------

```
import java.util.*;
public class Test3
{
public static void main(String[] args)
{
                ArrayList<String> al = new ArrayList<>();  //Generic type
                al.add("Ravi");
                al.add("Ajay");
                al.add("Vijay");

      for(int i=0; i<al.size(); i++)
                {
                String name = al.get(i); //no type casting is required
                System.out.println(name.toUpperCase());
                }
```

```
   }
}
```

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

```
//Program that describes the return type of any method can be type safe
//[We can apply generics on method return type]

import java.util.*;
public class Test4
{
        public static void main(String [] args)
        {
                Dog d1 = new Dog();
                Dog d2 = d1.getDogList().get(1);
                System.out.println(d2);
        }
}
class Dog
{
        public List<Dog> getDogList()
        {
                List<Dog> d = new ArrayList<>();
                d.add(new Dog());
                d.add(new Dog());
                return d;
        }
}
```

**Note**:- In the above program the compiler will stop us from returning anything which is not compaitable List<Dog> and there is a gurantee that only "type safe list of Dog object" will be returned so we need not to provide type casting as shown below

```
Dog d2 = (Dog) d1.getDogList().get(0);  //before generic.
```

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

```
//Mixing generic with non-generic

import java.util.*;
class Car
{
}
public class Test5
{
        public static void main(String [] args)
        {
        ArrayList<Car> a = new ArrayList<>();
        a.add(new Car());
        a.add(new Car());
   a.add(new Car());

        ArrayList b = a;  //assigning Generic to raw type

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

```
		}
}
```

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

```
//Mixing generic to non-generic
import java.util.*;
public class Test6
{
	public static void main(String[] args)
	{
		List<Integer> myList = new ArrayList<>();
		myList.add(4);
		myList.add(6);
		myList.add(5);

		UnknownClass u = new UnknownClass();
		int total = u.addValues(myList);
		System.out.println("The sum of Integer Object is :"+total);
	}
}
class UnknownClass
{
	public int addValues(List list)  //safe Object to unsafe object OR generic to raw type
	{
	Iterator it = list.iterator();
	int total = 0;
	while (it.hasNext())
	{
		int i = ((Integer)it.next());
		total += i;			//total = 15
	}
	return total;
	}
}
```

**Note**:-

In the above program the compiler will not generate any warning message because even though we are assigning type safe Integer Object to unsafe or raw type List Object but this List Object is not inserting anything new in the collection so there is no risk to the caller.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

```
//Mixing generic to non-generic

import java.util.*;
public class Test7
{
	public static void main(String[] args)
	{
		List<Integer> myList = new ArrayList<>();

		myList.add(4);
```

```
                myList.add(6);
                UnknownClass u = new UnknownClass();
                int total = u.addValues(myList);
                System.out.println(total);
        }
}
class UnknownClass
{
  public int addValues(List list)
        {
                list.add(5);        //adding object to raw type
                Iterator it = list.iterator();
                int total = 0;
                while (it.hasNext())
                {
                int i = ((Integer)it.next());
                total += i;
                }
                return total;
        }
}
```

---

**Type Erasure**

In the above program the compiler will generate warning message because the unsafe List Object is inserting the Integer object 5 so the type safe Integer object is getting value 5 from unsafe type so there is a problem to the caller method.

By writing ArrayList<Integer> actually JVM does not have any idea that our ArrayList was suppose to hold only Integers.

All the type safe information does not exist at runtime. All our generic code is Strictly for compiler. There is a process done by java compiler called "Type erasure" in which the java compiler converts generic version to non-generic type.

List<Integer> myList = new ArrayList<Integer>();

At the compilation time it is fine but at runtime for JVM the code becomes

List myList = new ArrayList();

**Note**:- GENERIC IS STRICTLY A COMPILE TIME PROTECTION.

---

Behavior of Polymorphism with Array and Generics.

---

```
//Polymorphism with array

import java.util.*;
abstract class Animal
{
        public abstract void checkup();
}

class Dog extends Animal
{
        @Override
        public void checkup()
        {
                System.out.println("Dog checkup");
        }
}

class Cat extends Animal
{
        @Override
        public void checkup()
        {
                System.out.println("Cat checkup");
        }
}

class Bird extends Animal
{
        @Override
        public void checkup()
        {
                System.out.println("Bird checkup");
        }
}

public class  Test8
{
        public void checkAnimals(Animal animals[]) //
        {
                for(Animal animal : animals)
                {
                        animal.checkup();
                }
        }

        public static void main(String[] args)
        {
                Dog []dogs={new Dog(), new Dog()};

                Cat []cats={new Cat(), new Cat(), new Cat()};

                Bird []birds = {new Bird(), new Bird()};

                Test8 t = new Test8();

                t.checkAnimals(dogs);
                t.checkAnimals(cats);
                t.checkAnimals(birds);
        }
}
```
**Note**:-From the above program it is clear that polymorphism(Upcasting) concept works with array.

---

```
import java.util.*;
abstract class Animal
{
        public abstract void checkup();
}

class Dog extends Animal
{
   @Override
        public void checkup()
        {
                System.out.println("Dog checkup");
        }
}

class Cat extends Animal
{
        @Override
        public void checkup()
        {
                System.out.println("Cat checkup");
        }
}
class Bird extends Animal
{
        @Override
        public void checkup()
        {
                System.out.println("Bird checkup");
        }
}
public class Test9
{
        public void checkAnimals(List<Animal> animals)
        {
                for(Animal animal : animals)
                {
       animal.checkup();
                }
        }
        public static void main(String[] args)
        {
                List<Dog> dogs = new ArrayList<>();
                dogs.add(new Dog());
```

```
                    dogs.add(new Dog());

                    List<Cat> cats = new ArrayList<>();
                    cats.add(new Cat());
                    cats.add(new Cat());

                    List<Bird> birds = new ArrayList<>();
                    birds.add(new Bird());

                    Test9 t = new Test9();
                    t.checkAnimals(dogs);
                    t.checkAnimals(cats);
                    t.checkAnimals(birds);
          }}
```

**Note**:- The above program will generate the compilation error.So from the above program it is clear that polymorphism does not work in the same way for generics as it does with arrays.

Eg:-

```
Parent [] arr = new Child[5]; //valid
Object [] arr = new String[5]; //valid
```

But in generics the same type is not valid

```
List<Object> list = new ArrayList<Integer>(); //Invalid
List<Parent> mylist = new ArrayList<Child>(); //Invalid
```

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

```
public class Test10
{
public static void main(String [] args)
          {
                    Object []obj = new String[3]; //valid with Array
                    obj[0] = "Ravi";
                    obj[1] = "hyd";
                    obj[2] = 12; //java.lang.ArrayStoreException
          }
}
```

**Note** :- It will generate java.lang.ArrayStoreException because we are trying to insert 12 (integer value) into String array.

In Array we have an Exception called ArrayStoreException but the same Exception or such type of exception, is not available with Generics that is the reason in generics compiler does not allow upcasting concept.

(It is a strict compile time checking)

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

```
import java.util.*;
class Parent
{
}
class Child extends Parent
{
}

public class Test11
{
public static void main(String [] args)
        {
                //ArrayList<Parent> lp = new ArrayList<Child>();

                ArrayList<Parent> lp1 = new ArrayList<Parent>();

                ArrayList<Child> lp2 = new ArrayList<>();

                System.out.println("Success");
        }
}
```

---

**Wild card character(?):**

- <?>          -: Many possibilities

- <Animal>          -: Only <Animal> can assign, but not Dog          or sub type of animal

- <? super Dog>   -: Dog, Animal, Object can assign (Compiler has          surity)

- <? extends Animal> -: Below of Animal(Child of Animal) means, sub classes of Animal (But the compiler does not have surity because you can have many sub classes of Animal in the future, so chances of wrong collections)

---

```
//program on wild-card chracter

import java.util.*;
class Parent
{

}
class Child extends Parent
{
}
public class Test12
{
public static void main(String [] args)
```

```
          {
                    List<?> lp = new ArrayList<Parent>();
                    System.out.println("Wild card ...");
          }
}
```

-------------------------------------------------------------------

```
import java.util.*;
public class Test13
{
          public static void main(String[] args)
          {
                    List<? extends Number> list1 = new ArrayList<Integer>();

                    List<? super String> list2 = new ArrayList<Object>();

                    List<Integer> list3 = new ArrayList<Integer>();

                    List list4 = new ArrayList();

                    System.out.println("yes");
          }
}
```

-------------------------------------------------------------------

```
import java.util.*;
public class  Test14
{
          public static void main(String[] args)
          {
                    try
                    {
          List<Object> x = new ArrayList<>(); //Array of Object[java 9]
        x.add(10);
                              x.add("Ravi");
                              x.add(true);
                              x.add(34.89);
                              System.out.println(x);
                    }
                    catch (Exception e)
                    {
                              System.out.println(e);
                    }
          }
}
```

-------------------------------------------------------------------

```
class MyClass<T>
{
          T obj;
          public MyClass(T obj)      //obj = 12 that is Integer Object
          {
                    this.obj=obj;
          }
```

```
        T getObj()
        {
                return obj;
        }
}
public class Test15
{
        public static void main(String[] args)
        {
                Integer i=12;
                MyClass<Integer> mi = new MyClass<Integer>(i);
                System.out.println("Integer object stored :"+mi.getObj());

                Float f=12.34f;
                MyClass<Float> mf = new MyClass<Float>(f);
                System.out.println("Float object stored :"+mf.getObj());

                MyClass<String> ms = new MyClass<String>("Rahul");
                System.out.println("String object stored :"+ms.getObj());

                MyClass<Boolean> mb = new MyClass<Boolean>(false);
                System.out.println("Boolean object stored :"+mb.getObj());

                Double d=99.34;
                MyClass<Double> md = new MyClass<Double>(d);
                System.out.println("Double object stored :"+md.getObj());

        }
}
-------------------------------------------------------------

//E stands for Element type
class Fruit
{
}
class Apple extends Fruit  //Fruit is the super, Apple is sub class
{
}


class Basket<E>     //E type is Fruit
{
        private E element;
        public void setElement(E element) //Fruit element = new Apple();
        {
                this.element = element;
        }

        public E getElement()
        {
                return this.element;
```

```
	}
}

public class Test16
{
	public static void main(String[] args)
	{
		Basket<Fruit> b = new Basket<Fruit>();
		b.setElement(new Apple());


		Apple x = (Apple)b.getElement();
		System.out.println(x);

	}}
```

-----------------------------------------------------------

**Concurrent collections in java**

Concurrent Collections are introduced from JDK 1.5 onwards to enhance the performance of multithreaded application.

These are threadsafe collection and available in java.util.concurrent sub package.

**Limitation of Traditional Collection:**

1) In the Collection framework most of the Collection classes are not thread-safe because those are non-synchronized like ArrayList, LinkedList, HashSet, HashMap is non-synchronized in nature, So If multiple threads will perform any operation on the collection object simultaneously then we will get some wrong data this is known as Data race or Race condition.

2) Some Collection classes are synchronized like Vector, Hashtable but performance wise these classes are slow in nature.

Collections class has provided static methods to make our List, Set and Map interface classes as a synchronized.

a) public static List synchronizedList(List list)

b) public static Set synchronizedSet(Set set)

c) public static Map synchronizedMap(Map map)

3) We can t modify the collection object data while iterating the Collection object. If we try to modify the collection, it throws ConcurrentModificationException, So non-synchronized collections are not good choice for Multi-threaded applications.

-----------------------------------------------------------

Programs:

```
import java.util.*;
public class Collection1
{
    public static void main(String args[])
    {
        ArrayList al = new ArrayList();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);
                    al.add(50);
        System.out.println("Arraylist Elements : "+al);
        Set s = new HashSet(al);
        System.out.println("Set Elements are: "+s);
    }
}
```

--------------------------------------------------------------------

```
//Collections.synchronizedList(List list);
import java.util.*;
public class Collection2
{
            public static void main(String[] args)
            {
                        ArrayList<String> arl = new ArrayList<>();
                        arl.add("Apple");
                        arl.add("Orange");
                        arl.add("Grapes");
                        arl.add("Mango");
                        arl.add("Guava");
                        arl.add("Mango");

                        List<String> syncCollection = Collections.synchronizedList(arl);

                        List<String> upperList = new ArrayList<>(); //New List

                        Runnable listOperations = () ->
                        {
                                    synchronized (syncCollection)
                                    {
            syncCollection.forEach(str -> upperList.add(str.toUpperCase()));
        }
    };

            Thread t1 = new Thread(listOperations);
            t1.start();

     upperList.forEach(x -> System.out.println(x));
            }
}
```

--------------------------------------------------------------------

```
//Collections.synchronizedSet(Set set);
import java.util.*;
public class Collection3
```

```
{
   public static void main(String[] args)
                     {
      Set<String> set = Collections.synchronizedSet(new HashSet<>());
      set.add("Apple");
                     set.add("Orange");
                     set.add("Grapes");
                     set.add("Mango");
                     set.add("Guava");
                     set.add("Mango");
      System.out.println("Set after Synchronization :");
      synchronized (set)
                     {
        Spliterator<String> itr = set.spliterator();
                       itr.forEachRemaining(str -> System.out.println(str));
       }
    }
}
---------------------------------------------------------------
//Collections.synchronizedMap(Map map);
import java.util.*;
public class Collection4
{
   public static void main(String[] args)
            {
      Map<String, String> map = new HashMap<String, String>();
      map.put("1", "Ravi");
      map.put("4", "Elina");
      map.put("3", "Aryan");
      Map<String, String> synmap = Collections.synchronizedMap(map);
      System.out.println("Synchronized map is :" + synmap);
    }
}
---------------------------------------------------------------
import java.util.*;
class ConcurrentModification extends Thread
{
            ArrayList<String> al = null;
            public ConcurrentModification(ArrayList<String> al) //al = arl
            {
                        this.al = al;
            }

            @Override
            public void run()
            {
                        try
                        {
                                    Thread.sleep(1000);
                        }

                        catch (InterruptedException e)
```

```
                {
                }
                al.add("KIWI");
        }
}
public class Collection5
{
        public static void main(String[] args) throws InterruptedException
        {
                ArrayList<String> arl = new ArrayList<>();
                arl.add("Apple");
                arl.add("Orange");
                arl.add("Grapes");
                arl.add("Mango");
                arl.add("Guava");

                ConcurrentModification cm = new ConcurrentModification(arl);
                cm.start();

                Iterator<String> itr = arl.iterator();
    while(itr.hasNext())
                {
                        String str = itr.next();
                        System.out.println(str);
                        Thread.sleep(1500);
                }
        }
}
```

**Note**:- The above program will throw an exception i.e ConcurrentModificationException because ArrayList class is non-synchronized so, it cannot handle multiple threads

**CopyOnWriteArrayList in java**:

- public class CopyOnWriteArrayList implements List, Cloneable, Serializable
- A CopyOnWriteArrayList is similar to an ArrayList but it has some additional features like thread-safe. This class is existing in java.util.concurrent sub package.
- ArrayList is not thread-safe. We can t use ArrayList in the multi-threaded environment because it creates a problem in ArrayList values  (Data inconsistency).
- The CopyOnWriteArrayList is an enhanced version of ArrayList. If we are making any modifications(add, remove, etc.)  in  CopyOnWriteArrayList then JVM creates a new copy by use of Cloning.
- The CopyOnWriteArrayList is costly, if we want to perform update operations, because whenever we make any changes the JVM creates a cloned copy of the array and add/update element to it.
- It is a thread-safe version of ArrayList. Multiple threads can read the data but only one thread can write the data at one time.
- CopyOnWriteArrayList is the best choice if we want to perform read operation frequently in multithreaded environment.
- The CopyOnWriteArrayList is a replacement of a synchronized List, because it offers better concurrency.

**Constructors of CopyOnWriteArrayList in java:**

We have 3 constructors :

1) CopyOnWriteArrayList c = new CopyOnWriteArrayList();

It creates an empty list in memory. This constructor is useful when we want to create a list without any value.

2) CopyOnWriteArrayList c = new CopyOnWriteArrayList(Collection c);

Interconversion of collections.

3) CopyOnWriteArrayList c = new CopyOnWriteArrayList(Object[] obj) ;

It Creates a list that containing all the elements that is specified Array. This constructor is useful when we want to create a CopyOnWriteArrayList from Array.

---

```
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

public class CopyOnWriteArrayListExample1
{
   public static void main(String[] args)
   {
      List<String> list = Arrays.asList("Apple", "Orange", "Mango","Kiwi", "Grapes");


      CopyOnWriteArrayList<String> copyOnWriteList = new CopyOnWriteArrayList<String>(list);

      System.out.println("Without modification = "+copyOnWriteList);

                        //Iterator1
      Iterator<String> iterator1 = copyOnWriteList.iterator();

      //Add one element and verify list is updated
      copyOnWriteList.add("Guava");

      System.out.println("After modification = "+copyOnWriteList);

      //Iterator2
      Iterator<String> iterator2 = copyOnWriteList.iterator();

      System.out.println("Element from first Iterator:");
      iterator1.forEachRemaining(System.out :: println);

      System.out.println("Element from Second Iterator:");
      iterator2.forEachRemaining(System.out :: println);
   }
}
```

---

```
import java.util.*;
import java.util.concurrent.*;
class ConcurrentModification extends Thread
{
        CopyOnWriteArrayList<String> al = null;
        public ConcurrentModification(CopyOnWriteArrayList<String> al)
        {
                this.al = al;
        }
        @Override
        public void run()
        {
                try
                {
                        Thread.sleep(1000);
                }

                catch (InterruptedException e)
                {
                }
          al.add("KIWI");
        }
}
public class CopyOnWriteArrayListExample2
{
        public static void main(String[] args) throws InterruptedException
        {
                CopyOnWriteArrayList<String> arl = new CopyOnWriteArrayList<>();
                arl.add("Apple");
                arl.add("Orange");
                arl.add("Grapes");
                arl.add("Mango");
                arl.add("Guava");
                ConcurrentModification cm = new ConcurrentModification(arl);
                cm.start();

                Iterator<String> itr = arl.iterator();
    while(itr.hasNext())
                {
                        String str = itr.next();
                        System.out.println(str);
                  Thread.sleep(1500);
                }

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

                Spliterator<String> spl =  arl.spliterator();
                spl.forEachRemaining(x -> System.out.println(x));

}
```

---

```
            }




}
```

We can use CopyOnWriteArrayList rather than the ArrayList for following cases

**1. Thread safe**

The CopyOnWriteArrayList is thread safe version of ArrayList. If we want to work in multithread environment then we should choose the CopyOnWriteArrayList. Because we can perform read and write operation by multiple threads at a time.

*Reason*: Because ArrayList is not thread safe. In ArrayList different threads make different changes that leads to inaccuracy of data. But in case of CopyOnWriteArrayList, whenever any thread makes changes in the original list, the JVM creates a new copy of list internally. So that value is updated for the all threads.

**Note**: In CopyOnWriteArrayList only one thread can perform write operation at a time But multiple threads can perform read operations simultaneously

**2. Minimum modification and frequently read operation**

As we already know CopyOnWriteArrayList is very costly because whenever we make any modification in list it creates a fresh new copy of list. So, we should use the CopyOnWriteArrayList when the number of write operations is very small as compared to the read operations.

*Reason*: For example we are creating a list from CopyOnWriteArrayList that contains 1000 objects.

CopyOnWriteArrayList<String> copyOnWriteList = new CopyOnWriteArrayList<String>(listOfthousands);

Now we are adding three more elements in list.

copyOnWriteList.add("Ravi");

copyOnWriteList.add("Ankit");

copyOnWriteList.add("ELina");

Each time, The JVM creates a new fresh copy of list. It means JVM creates three new copies for three operations. Now we can think if we are doing number of modification in list it will be very costly.

---

**CopyOnWriteArraySet :**

- public class CopyOnWriteArraySet extends AbstractSet implements Serializable
- A CopyOnWriteArraySet is a thread-safe version of HashSet in Java and it works like CopyOnWriteArrayList in java.
- The CopyOnWriteArraySet internally used CopyOnWriteArrayList to perform all type of operation.It means the CopyOnWriteArraySet internally creates an object of CopyOnWriteArrayList and perform operation on it.
- Whenever we perform add, set, and remove operation on CopyOnWriteArraySet, it internally creates a new object of CopyOnWriteArrayList and copies all the data to the new object. So, when it is used in by multiple threads, it doesn t create a problem, but it is well suited if we have small size collection and want to perform only read operation by multiple threads.
- The CopyOnWriteArraySet is the replacement of synchronizedSet and offers better concurrency.

- It creates a new copy of the array every time iterator is created, so performance is slower than HashSet.

**Constructors:**

It has two constructors

1)CopyOnWriteArraySet set1 = new CopyOnWriteArraySet();

It will create an empty Set.

2) CopyOnWriteArraySet set1 = new CopyOnWriteArraySet(Collection c); Interconversion of collection.

---

```
import java.util.ArrayList;
import java.util.concurrent.CopyOnWriteArraySet;

public class CopyOnWriteArraySetExample1
{
   public static void main(String[] args)
   {
      ArrayList<String> list = new ArrayList<String>();
      list.add("Apple");
      list.add("Orange");
      list.add("Grapes");
                  list.add("Grapes");

      CopyOnWriteArraySet<String> set = new CopyOnWriteArraySet<String>(list);
      System.out.println("Element from Set: "+ set);
   }
}
```

---

```
import java.util.concurrent.CopyOnWriteArraySet;

public class CopyOnWriteArraySetExample2
{
   public static void main(String[] args)
   {
      CopyOnWriteArraySet<Integer> set = new CopyOnWriteArraySet<Integer>();
      set.add(1);
      set.add(2);
      set.add(3);
      set.add(4);
      set.add(5);

      System.out.println("Is element contains: "+set.contains(1));

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

      System.out.println("remove element from set: "+set.remove(3));

      System.out.println("Element from Set: "+ set);
```

```
    }
}
```

-------------------------------------------------------------------

**ConcurrentHashMap:**

- public class ConcurrentHashMap<K,V> extends AbstractMap<K,V> implements ConcurrentMap<K,V>, Serializable
- Like HashMap, ConcurrentHashMap provides similar functionality except that it has internally maintained concurrency.
- It is the concurrent version of the HashMap. It internally maintains a Hashtable that is divided into segments(Buckets).
- The number of segments depends upon the level of concurrency required the Concurrent HashMap. By default, it divides into 16 segments and each Segment behaves independently. It doesn t lock the whole HashMap as done in Hashtables/synchronizedMap, it only locks the particular segment(Bucket) of HashMap. [Bucket level locking]
- ConcurrentHashMap allows multiple threads can perform read operation without locking the ConcurrentHashMap object.
- It does not allow null as a key or evan as a value.

*[Note :- TreeSet, TreeMap, Hashtable, PriroityQueue, ConcurrentHashMap]*

**It contains 5 types of constructors:**

1) ConcurrentHashMap chm1 = new ConcurrentHashMap();

2) ConcurrentHashMap chm2 = new ConcurrentHashMap(int initialCapacity);

3) ConcurrentHashMap chm3 = new ConcurrentHashMap(int initialCapacity, float loadFactor);

4) ConcurrentHashMap chm4 = new ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel);

5) ConcurrentHashMap chm5 = new ConcurrentHashMap(ConcurrentMap m);

**Internal Working of ConcurrentHashMap**:

- Like HashMap and Hashtable, the ConcurrentHashMap is also used Hashtable data structure. But it is using the segment locking strategy to handle the multiple threads. A segment(bucket) is a portion of ConcurrentHashMap and ConcurrentHashMap uses a separate lock for each thread. Unlike Hashtable or synchronized HashMap, it doesn t synchronize the whole HashMap or Hashtable for one thread.
- As we have seen in the internal implementation of the HashMap, the default size of HashMap is 16 and it means there are 16 buckets. The ConcurrentHashMap uses the same concept is used in ConcurrentHashMap. It uses the 16 separate locks for 16 buckets by default because the default concurrency level is 16. It means a ConcurrentHashMap can be used by 16 threads at same time. If one thread is reading from one bucket(Segment), then the second bucket doesn t affect it.

*Why we need ConcurrentHashMap in java?*

As we know Hashtable and HashMap works based on key-value pairs. But why we are introducing another Map? As we know HashMap is not thread safe, but we can make it thread-safe by using Collections.synchronizedMap() method and Hashtable is thread-safe by default.

But a synchronized HashMap or Hashtable is accessible only by one thread at a time because the object get the lock for the whole HashMap or Hashtable. Even multiple threads can t perform read operations at the same time. It is the main disadvantage of Synchronized HashMap or Hashtable, which creates performance issues. So ConcurrentHashMap provides better performance than Synchronized HashMap or Hashtable.

---

```
import java.util.HashMap;
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample1
{
  public static void main(String args[])
  {

      HashMap<Integer, String> hashMap = new HashMap<Integer, String>();
      hashMap.put(1, "Ravi");
      hashMap.put(2, "Ankit");
      hashMap.put(3, "Prashant");
            hashMap.put(4, "Pallavi");

      ConcurrentHashMap<Integer, String> concurrentHashMap = new ConcurrentHashMap<Integer,
String>(hashMap);
      System.out.println("Object from ConcurrentHashMap: "+ concurrentHashMap);

  }

}
```

---

```
import java.util.Iterator;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample2
{
  public static void main(String args[])
  {
      // Creating ConcurrentHashMap
      Map<String, String> cityTemperatureMap = new ConcurrentHashMap<String, String>();

       cityTemperatureMap.put("Delhi", "30");
       cityTemperatureMap.put("Mumbai", "32");
       cityTemperatureMap.put("Chennai", "35");
       cityTemperatureMap.put("Bangalore", "22" );

       Iterator<String> iterator = cityTemperatureMap.keySet().iterator();
```

```
        while (iterator.hasNext())
        {
          System.out.println(cityTemperatureMap.get(iterator.next()));
          // adding new value, it won't throw error
          cityTemperatureMap.put("Hyderabad", "28");
        }
    }
```