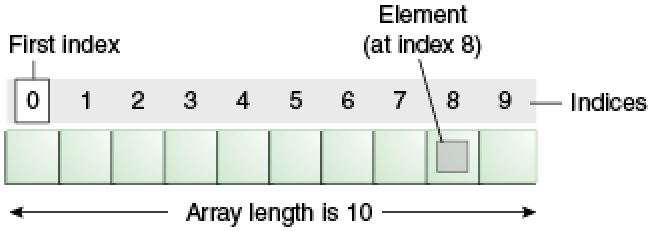
Array-

- An array is a collection of similar type of elements which has contiguous memory location.
- Contiguous memory = Continuous block of memory used to store array elements side by side
- Java array is an object which contains elements of a similar data type.
- We can store only a fixed set of elements in a Java array.
- Array in Java is index-based, the first element of the array is stored at the 0th index, and 2nd element is stored on 1st index and so on.



Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages:

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.
- Readymade or predefined method support is not available. We have to write the code for it.
- It is homogenous in nature means can store only one type of data.

There are two types of array as:

- Single Dimensional Array
- Multidimensional Array

- **Syntax to Declare an Array in Java**
- dataType[] arr;

- **Instantiation of an Array in Java**
- arr = new datatype[size];

Syntax for Multidimensional Array as:

data_type[1st dimension][2nd dimension][]..[Nth dimension] **array_name** = **new data_type**[size1][size2]….[sizeN];

Where:

- **data_type**: Type of data to be stored in the array. For example: int, char, etc.
- **dimension**: The dimension of the array created. For example: 1D, 2D, etc.
- **array_name**: Name of the array
- **size1, size2, …, sizeN**: Sizes of the dimensions respectively.

Two dimensional array:

int[][] twoD_arr = new int[10][20];

Three dimensional array:

int[][][] threeD_arr = new int[10][20][30];

Java Program to illustrate how to declare, instantiate, initialize and traverse the Java array.

```
public class ArrayTest {

      // Collection of homogenous data type is called an array

      public static void main(String[] args) {

           int [] intarray = new int [5]; //declaration and instantiation


           intarray [0]=5;   //initialization
           intarray [1]=10;
           intarray [2]=15;
           intarray [3]=20;
           intarray [4]=25;

         int intsize = intarray.length;//length is the property of array

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

```
                   //traversing array
                  for (int i=0; i<intsize; i++) {

                     System.out.println(intarray[i]);
                  }

             }
       }
```

Output:

```
5
5
10
15
20
25
```

- We can declare, instantiate and initialize the java array together by:
- int a[]={10,20,30};//declaration, instantiation and initialization

```
public class AverageOfElements {

       public static void main(String[] args) {

             int a [] = {10 , 26 , 29 , 34, 76, 49 , 53};

             int sum =0 ;

             for (int i=0 ; i<a.length ; i++) //declaration, instantiation and 
initialization
             {


                    sum = sum + a[i];
             }

             System.out.println("Sum of all elements as :" + sum);
       }
}
```

Output:

```
Sum of all elements as :277
```

- For-each Loop for Java Array
- We can also print the Java array using for-each loop. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.
- The syntax of the for-each loop is given below:

```
for(data_type variable:array){
//body of the loop
}
```

Java Program to print the array elements using for-each loop:

```
public class Test3 {

            public static void main(String[] args) {

             int [] a = new int [7];

             a [0]=10;
             a [1]=20;
             a [2]=30;

             a [3]=40;
             a [4]=50;
             a [5]=60;
             a [6]=70;

             int size =a.length;

             for (int aa:a) {

              System.out.println("Iterate value are: "+ aa);
                   }

            }

}
```

Output:

```
Iterate value are: 10
Iterate value are: 20
Iterate value are: 30
Iterate value are: 40
Iterate value are: 50
Iterate value are: 60
Iterate value are: 70
```

Java Program to Sort the array elements using for-each loop:

```
public class ArrayOrder {

      public static void main(String[] args) {

       int [] jk = new int [5] ;


           jk [0]=3;
           jk [1]=8;
           jk [2]=5;
           jk [3]=11;
           jk [4]=15;

           System.out.println("**********Before Sorting of array********");

            int size = jk.length;
            System.out.println(size);
            for (int bb:jk) {

               System.out.println(bb);
            }

            System.out.println("**********AfterSortingOfArray************");

            Arrays.sort(jk);

            for (int cx:jk) {

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

Output:

```
**********Before Sorting of array********
5
3
8
5
11
15
**********AfterSortingOfArray************
3
5
8
11
15
```

If we initialize data beyond size of array then it will throws an exceptions.

```
public class ArrayTest {

	public static void main(String[] args) {

		int [] intarray = new int [5]; //declaration and 
instantiation


		intarray [0]=5;   //initialization
		intarray [1]=10;
		intarray [2]=15;
		intarray [3]=20;
		intarray [5]=25;  //trying to initialize 5th  index value

	       int intsize = intarray.length;      //length is the property 
of array

			System.out.println(intsize);


		 //traversing array
		for (int i=0; i<intsize; i++) {

		   System.out.println(intarray[i]);
		}

	}
}
```

Output :

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of 
bounds for length 5
	at Array.ArrayTest.main(ArrayTest.java:16)
```

Need of collection:

- Array are fixed in size, once it is created we cannot change its size based on requirements.
- Array can hold homogenous data elements.
- There is no readymade method support available in array.

Collection:

- Collection is a group of individual objects which are represented by single entity.

Limitation of Object [] array:

- Arrays are fixed in size that is once we created an array there is no chance of increasing (or) decreasing the size based on our requirement hence to use arrays concept compulsory we should know the size in advance which may not possible always.
- Arrays can hold only homogeneous data elements.

Example:

```
Student [] s=new Student [10000];
s[0]=new Student ();//valid

s[1]=new Customer ();//invalid(compile time error)
Compile time error:
```

Compile time error:

```
Test.java:7: cannot find symbol
Symbol: class Customer
Location: class Test
          s[1]=new Customer();
```

- **But we can resolve this problem by using object type array(Object[]).**
  Example:

```
  Object[] o=new Object[10000];
o[0]=new Student();
     o[1]=new Customer();
```

- **Arrays concept is not implemented based on some data structure hence ready-made methods support we can't expert. For every requirement we have to write the code explicitly.**

To overcome the above limitations of Array we should go for collections concept.

- Collections are growable in nature that is based on our requirement we can increase (or) decrease the size hence memory point of view collections concept is recommended to use.
- Collections can hold both homogeneous and heterogeneous objects.
- Every collection class is implemented based on some standard data structure.
- Hence for every requirement ready-made method support is available being a programmer we can use these methods directly without writing the functionality on our own.

Explain the different between the Array and Collection?

Arrays	**Collection**
Arrays are fixed in size.	Collections are growable in nature.
Memory point of view arrays are not recommended to use.	Memory point of view collections are highly recommended to use
Performance point of view arrays are recommended to use.	Performance point of view collections are not recommended to use.
Arrays can hold only homogeneous data type elements.	Collections can hold both homogeneous and heterogeneous elements.
There is no underlying data structure for arrays and hence there is no readymade method support.	Every collection class is implemented based on some standard data structure and hence readymade method support is available
Arrays can hold both primitives and object types.	Collections can hold only objects but not primitives.

Difference between Collection and Collections?

- Collection is an "interface" which can be used to represent a group of objects as a single entity.
- Whereas "Collections is a utility class" present in java.util package to define several utility methods for Collection objects.

Collection-------------------interface

Collections-----------------class

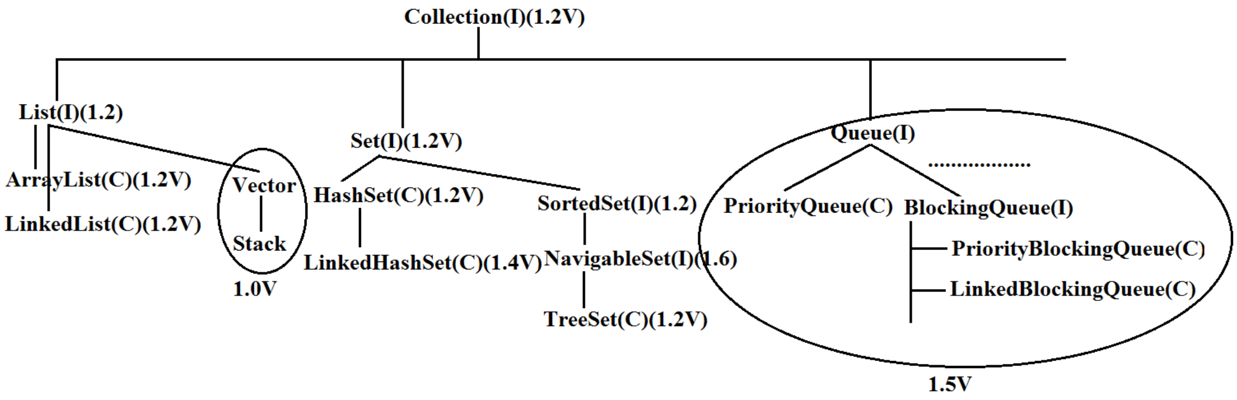
Collection Framework-

- Why it is called as framework because it contain the collection of classes and interface that work together.

- **9(Nine) key interfaces of collection framework:**
  - Collection
  - List
  - Set
  - SortedSet
  - Navigable Set
  - Queue
  - Map
  - Sorted Map
  - Navigable Map

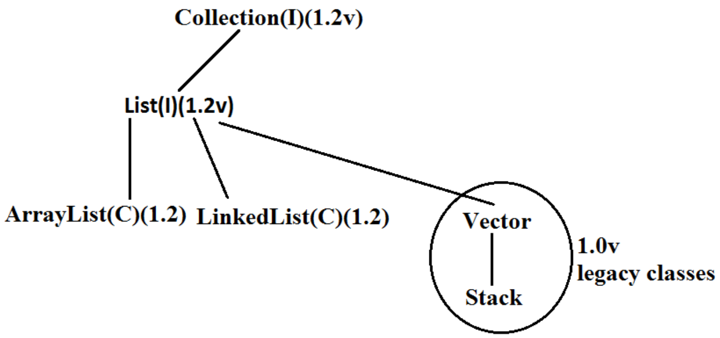
Collection

- If we want to represent a group of "individual objects" as a single entity then we should go for collection.
- In general we can consider collection as root interface of entire collection framework.
- Collection interface defines the most common methods which can be applicable for any collection object
- There is no concrete class which implements Collection interface directly.



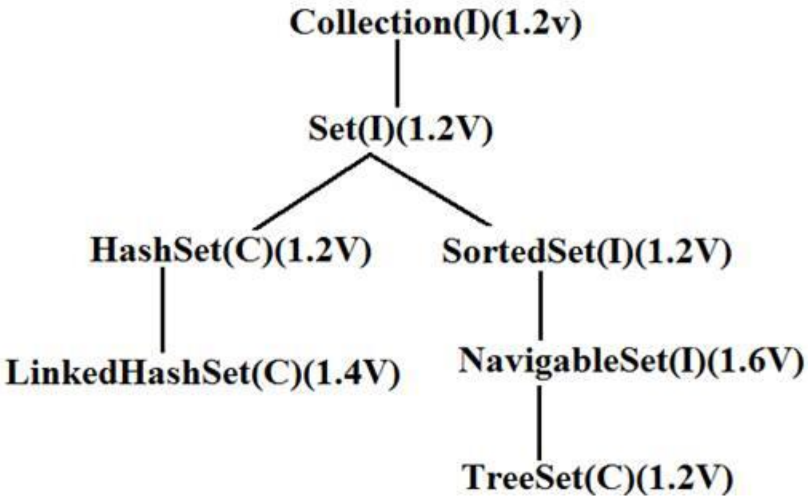
List

- It is child interface of collection.
- It is present in Java.util.Package.
- If we want to represent a group of individual objects as a single entity where **“duplicates are allow and insertion order must be preserved"** then we should go for List interface
- Vector and Stack classes are re-engineered in 1.2 versions to implement List interface.



Set

- It is child interface of collection.
- If we want to represent a group of individual objects as single entity **"where duplicates are not allow and insertion order is not preserved"** then we should go for Set interface.



SortedSet

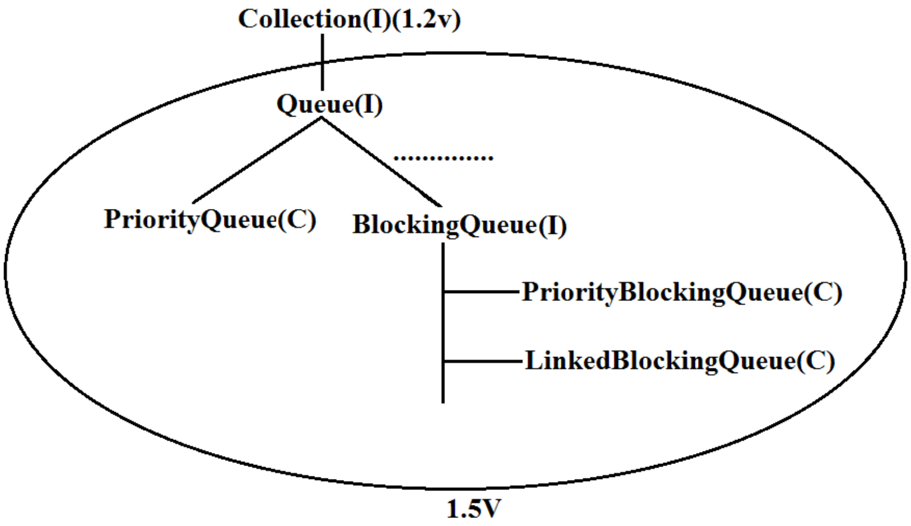
- It is the child interface of Set.
- If we want to represent a group of individual objects as single entity **"where duplicates are not allow but all objects will be insertion according to some sorting order then we should go for SortedSet.”**
- If we want to represent a group of **"unique objects"** according to some sorting order then we should go for SortedSet.

NavigableSet

- It is the child interface of SortedSet
- It provides several methods for navigation purposes

Queue:

- It is the child interface of Collection
- If we want to represent a group of individual objects prior to processing then we should go for queue concept.

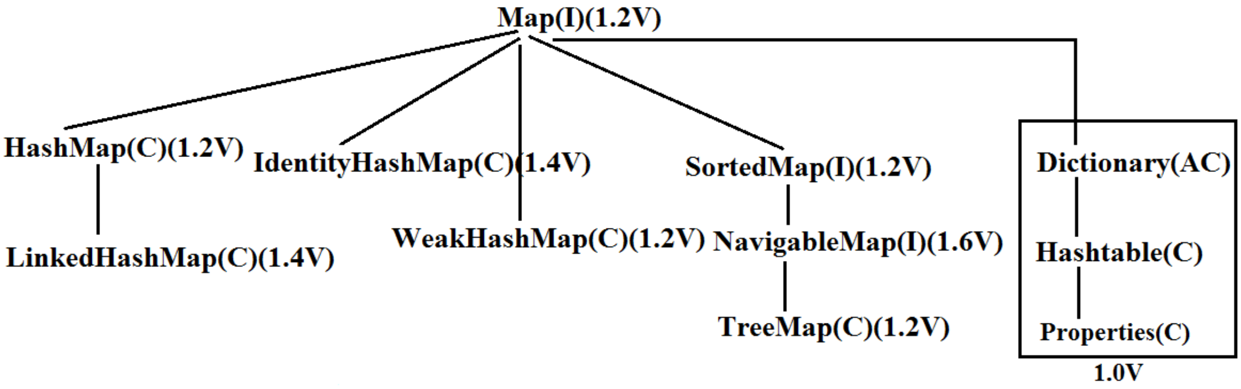


Note: All the above interfaces (Collection, List, Set, SortedSet, NavigableSet, and Queue) meant for representing a group of individual objects.

If we want to represent a group of objects as key-value pairs then we should go for Map.

▪ Map

- Map is not child interface of Collection.
- If we want to represent a group of objects as key-value pairs then we should go for Map interface.
- Duplicate keys are not allowed but values can be duplicated.



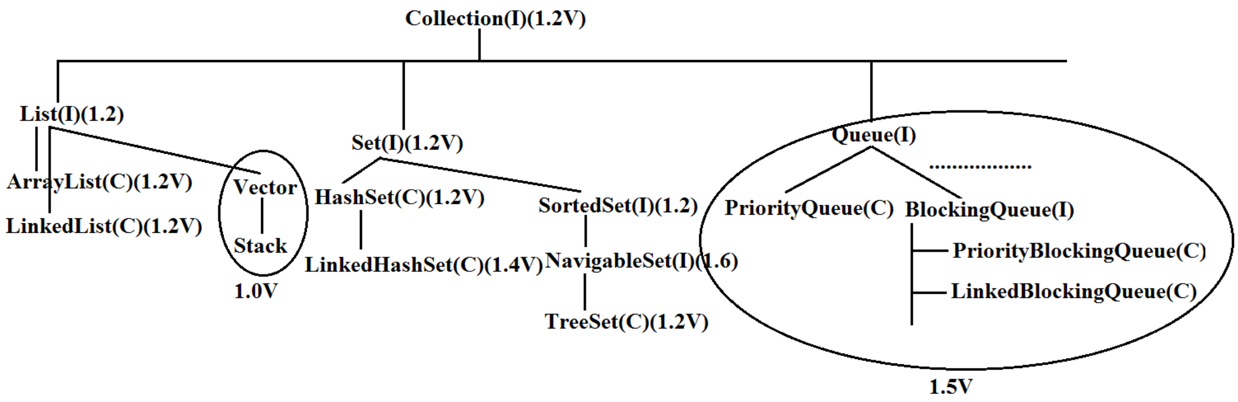
♦ SortedMap

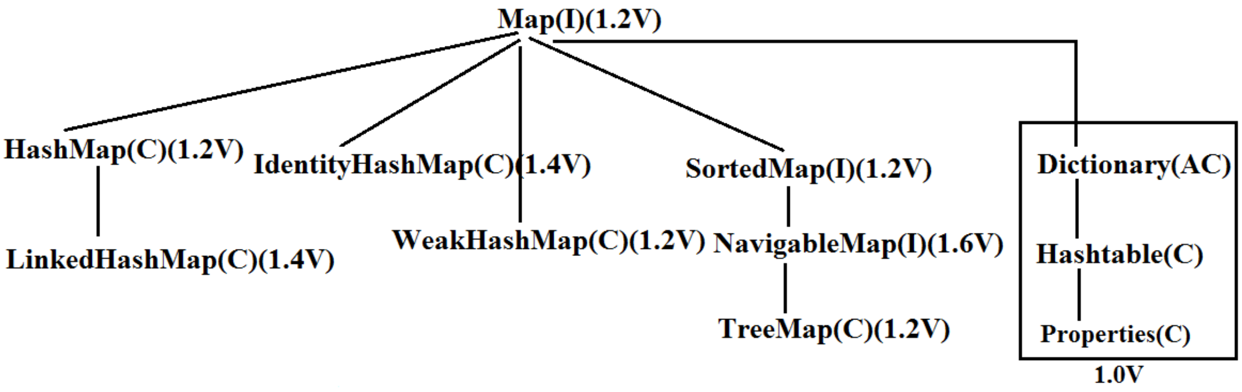
- It is the child interface of Map
- If we want to represent a group of objects as key value pairs **"according to some sorting order of keys"** then we should go for SortedMap

- **NavigableMap**
    - It is the child interface of SortedMap and defines several methods for navigation

In collection framework the following are legacy characters:

- **Enumeration(I)**
- **Dictionary(AC)**
- **Vector(C)**
- **Stack(C)**
- **Hashtable(C)**
- **Properties(C)**





Collection

- If we want to represent a group of "individual objects" as a single entity then we should go for collection.
- In general we can consider collection as root interface of entire collection framework.
- Collection interface defines the most common methods which can be applicable for any collection object
- The following is the list of methods present in Collection interface.

1. **boolean add(Object o);**
2. **boolean addAll(Collection c);**
3. **boolean remove(Object o);**
4. **boolean removeAll(Object o);**
5. **boolean retainAll(Collection c);**

- **To remove all objects except those present in c.**

6. **void clear();**
7. **boolean contains(Object o);**
8. **boolean containsAll(Collection c);**
9. **boolean isEmpty();**
10. **Int size();**
11. **Object[] toArray();**
12. **Iterator iterator();**

- There is no concrete class which implements Collection interface directly.

List

- It is child interface of collection.
- It is present in Java.util.Package.
- If we want to represent a group of individual objects as a single entity where **“duplicates are allow and insertion order must be preserved"** then we should go for List interface
- We can differentiate duplicate objects and we can maintain insertion order by means of index hence **"index play very important role in List".**
- Vector and Stack classes are re-engineered in 1.2 versions to implement List interface.
- **List interface defines the following specific methods**

1. **boolean add(int index, Object o);**
2. **boolean addAll(int index, Collection c);**
3. **Object get(int index);**
4. **Object remove(int index);**
5. **Object set(int index, Object new);//to replace**

6. Int indexOf(Object o);

Returns index of first occurrence of "o".

7. Int lastIndexOf(Object o);
8. ListIterator listIterator();

- ## Array List

- The underlying data structure is resizable array (or) growable array.
- Duplicates are allowed.
- Insertion order is preserved.
- Heterogeneous objects are allowed.
- Null insertion is possible. (We can add n number of null values in array list).
- **Incremental Capacity = (current capacity * 3/2) +1**
- Manipulation with Array list is slow because it internally uses an array. If any element is removed or added in the array, all the other bits are shifted in the memory. **So it's a worst choice for manipulation operation.**
- Best choice for retrieval operation.

- **Constructor-**
  1. **ArrayList al= new ArrayList();**

     Create the empty array list with default initial capacity 10. Once array list reaches its max capacity then new array list will be created with its new capacity.

     Incremental Capacity = (current capacity * 3/2) +1

  2. **ArrayList a=new ArrayList(int initialcapacity);**

     Creates an empty ArrayList object with the specified initial capacity.

  3. **ArrayList a=new ArrayList(collection c);**

  - Creates an equivalent ArrayList object for the given Collection that is this constructor meant for inter conversation between collection objects. Here we are passing **the objects.**

- **Java program on ArrayList with use of predefine Methods**

```
package collectionInterface;

import java.util.ArrayList;

public class ArrayList2 {

      public static void main(String[] args) {

               ArrayList jk = new ArrayList();

             //Add elements in arraylist.

               jk.add("Tanvir");

               jk.add("Shinde");

               jk.add(27);

               jk.add("Pune");

               jk.add(null);

               jk.add("Software tester");

               jk.add(null);

             System.out.println(jk);    //To print all the elements in arraylist

             System.out.println(jk.size());   //get the size

            System.out.println(jk.get(5));     //printing value at a particular index

          jk.set(4, "XYZ");
         System.out.println(jk);   //set a value at a particular index

           jk.remove(6);          //remove a value at a particular index
         System.out.println(jk);

       System.out.println(jk.contains("XYZ"));  //to check a value

       System.out.println(jk.isEmpty()); //to check if arraylit it is empty for not

      jk.clear();                    // to delete all records
       System.out.println(jk);

      }
}
```

Output on console:

```
[Tanvir, Shinde, 27, Pune, null, Software tester, null]
7
Software tester
[Tanvir, Shinde, 27, Pune, XYZ, Software tester, null]
[Tanvir, Shinde, 27, Pune, XYZ, Software tester]
true
false
[]
```

This is allowed but not recommended in Java. We are using Arraylist with Generics:

```
public class ArrayListDemo1 {

	public static void main(String[] args) {

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

		jk.add(10);  // add element

		jk.add(20);

		jk.add(30);

		ArrayList<Integer> jk2 = new ArrayList<Integer>();

		jk2.add(20);

		jk2.add(35);

		jk.addAll(jk2); //add elements of other arraylist

		System.out.println(jk);

		Iterator<Integer> ub = jk.iterator();
 // using iterator interface hasNext method we iterate values

		while (ub.hasNext()) {

			System.out.println(ub.next());
		}
	}
}
```

Output on Console:

```
[10, 20, 30, 20, 35]
10
20
30
20
35
```

ArrayList java program using for each loop:

```
import java.util.ArrayList;

public class ArrayListDemo2 {

	public static void main(String[] args) {

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

			jk.add(10);

			jk.add(20);

			jk.add(30);

			   for (  Integer js : jk) {

				  System.out.println(js);
			   }
			  System.out.println(jk);
	}
}
```

Output on Console:

```
10
20
30
[10, 20, 30]
```

Arraylist if we are using with Generics such as Integer, String and so on …This is recommended here only same type of Data.

- ArrayList is the best choice if our frequent operation is retrieval operation. It implements the random access interface.
- ArrayList is the worst choice if our frequent operation is insertion or deletion (because several shift operation are required for this).

- **To convert Array into Arraylist (Collection) and to convert the collection (Arraylist) into the Array.**

```
package collectionInterface;
import java.util.ArrayList;
import java.util.Arrays;
public class ArrayToCollection {

	public static void main(String[] args) {
     String [] jk = new String [4];

             jk[0] = "Tanvir";

             jk[1]=  "Atul";

             jk[2]="Shinde";

             jk[3]="Mishra";

             ArrayList<String> rt = new
ArrayList<String>(Arrays.asList(jk));

             System.out.println(rt);

             //to convert collection to array

             rt.add("123");

             rt.add("456");

             rt.add("789");

             System.out.println(rt.size());

             String [] str = new String [rt.size()];

             rt.toArray(str);

             for (String ss : str) {

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

Output on Console:

```
[Tanvir, Atul, Shinde, Mishra]
7
Tanvir
Atul
Shinde
Mishra
123
456
```

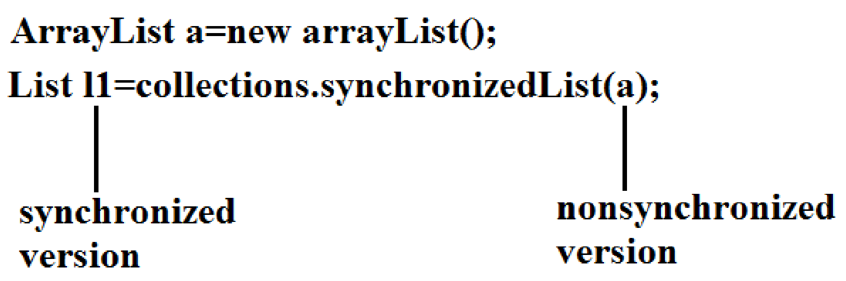
Difference between Arraylist and Vector?

ArrayList	Vector
No method is synchronized.	Every method is synchronized.
At a time multiple Threads are allow to operate on ArrayList object and hence ArrayList object is not Thread safe.	At a time only one Thread is allow to operate on Vector object and hence Vector Object is Thread safe.
Relatively performance is high because Threads are not required to wait.	Relatively performance is low because Threads are required to wait.
It is non legacy and introduced in 1.2v	It is legacy and introduced in 1.0v

Getting synchronized version of ArrayList object :

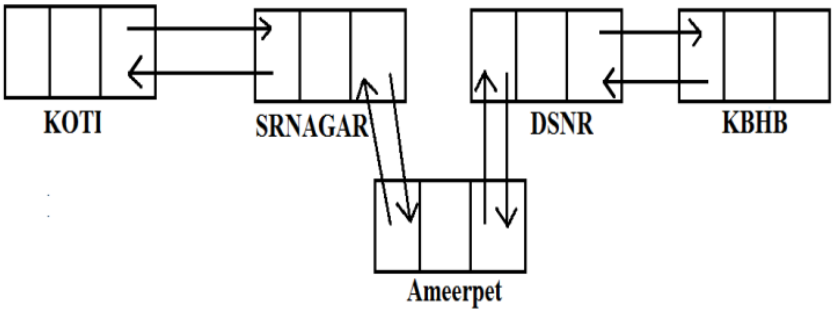
- Collections class defines the following method to return synchronized version of List.

Public static List synchronizedList(list l);



```
ArrayList a=new arrayList();
List l1=collections.synchronizedList(a);
```

- **Linked List**
  - The underlying data structure is double LinkedList.
  - If our frequent operation is insertion (or) deletion in the middle then LinkedList is the best choice.
  - If our frequent operation is retrieval operation then LinkedList is worst choice.
  - Insertion order is preserved.
  - Duplicates are allowed.
  - Heterogeneous objects are allowed.
  - Null insertion is possible.
  - LinkedList will implements serializable and clonable interface but not random access interface.
  - LinkedList is worst choice if our frequent operation is retrieval.



- Usually we can use LinkedList to implement Stacks and Queues.
- To provide support for this requirement LinkedList class defines the following 6 specific methods.
- We can apply these methods only on LinkedList object.

  void addFirst ();
  void addLast();
  Object getFirst();
  Object getLast();
  Object removeFirst();
  Object removeLast ();

- **Constructor:**
  1. **LinkedList l= new LinkedList();**
     Create the empty linked list object.

  2. **LinkedList ll= new LinkedList(Collection c);**
     To create an equivalent LinkedList object for the given collection.

- **Java Program on Linkedlist with some predefined methods of LinkList:**

```
package collectionInterface;

import java.util.LinkedList;

public class LinkedListDemo {

	public static void main(String[] args) {

		LinkedList linkedList = new LinkedList();

		linkedList.add(50);
		linkedList.add("Jeevan");
		linkedList.add(10);
		linkedList.add(null);
```

```
             System.out.println(linkedList);
             linkedList.addFirst("Pune");
             System.out.println(linkedList);
             linkedList.addLast("SoftwareTester");
             System.out.println(linkedList);
           System.out.println(linkedList.getFirst());
           System.out.println(linkedList.getLast());
           linkedList.removeFirst();
        System.out.println(linkedList);
        linkedList.removeLast();
        System.out.println(linkedList);
             }
}
```

Output on Console:

```
[50, Jeevan, 10, null]
[Pune, 50, Jeevan, 10, null]
[Pune, 50, Jeevan, 10, null, SoftwareTester]
Pune
SoftwareTester
[50, Jeevan, 10, null, SoftwareTester]
[50, Jeevan, 10, null]
```

```
public class LinkedListDemo2 {

      public static void main(String[] args) {

             LinkedList<Integer> linkedList = new LinkedList<Integer>();

                          linkedList.add(10);
                          linkedList.add(40);
                          linkedList.add(30);
                          linkedList.add(20);
                          linkedList.addFirst(70);
                          linkedList.addLast(80);

                          System.out.println(linkedList);

                          Iterator<Integer> ub = linkedList.iterator();

                          while (  ub.hasNext()) {

                                 System.out.println(ub.next());
                          }


                          for (   Integer df: linkedList )

                                 System.out.println(df);

      }
}
```

Output on Console:

```
[70, 10, 40, 30, 20, 80]
70
10
40
30
20
80
70
10
40
30
20
80
```

Difference between ArrayList and LinkedList:

ArrayList	**LinkedList**
Duplicates are allowed.	Duplicates are allowed.
Underlying data structure for ArrayList is resizable or growable array.	Underlying data structure is double linked list.
It is worst choice if our frequent operation is insertion and deletion.	It is the worst choice if our frequent operation is retrieval.
Default capacity of ArrayList is 10.	**Does not have a default capacity**.
Best choice for retrieval operation.	For retrieval of data Linked list is the worst choice
Allows any number of null values.	Allows any number of null values.
Order of insertion is maintained.	Order of insertion is maintained.
Arraylist is resizable. Incremental Capacity = (current capacity * 3/2) +1	**Does not have a default capacity**
Manipulation with Arraylist is slow because it internally uses an array. If any element is removed or added in the array, all the other bits are shifted in the memory. So it's a worst choice for manipulation operation.	For manipulation of data Linked list is the best choice as there is no Shifting of elements.

♦ Vector:

- The underlying data structure is resizable array (or) growable array.
- Duplicate objects are allowed.
- Insertion order is preserved.
- Heterogeneous objects are allowed.

- Null insertion is possible.
- Implements Serializable, Cloneable and Random-access interfaces.
- Every method present in Vector is synchronized and hence Vector is Thread safe

- **Vector specific methods:**

To add objects:

add (Object o); -----Collection
add (int index, Object o);-----List
addElement (Object o);-----Vector

To remove elements:

remove (Object o);--------Collection
remove (int index);--------------List
removeElement (Object o);----Vector
removeElementAt (int index);-----Vector
removeAllElements ();-----Vector
clear ();-------Collection

To get objects:

Object get(int index);---------------List
Object elementAt(int index);-----Vector
Object firstElement();--------------Vector
Object lastElement();---------------Vector

Other methods:

Int size(); //How many objects are added
Int capacity (); //Total capacity
Enumeration elements ();

Constructors:

1. Vector v=new Vector();

   Creates an empty Vector object with default initial capacity 10.
   Once Vector reaches its maximum capacity then a new Vector object will be created with double capacity.
   That is "newcapacity = currentcapacity*2".

2. Vector v=new Vector(int initialcapacity);
3. Vector v=new Vector(int initialcapacity, int incrementalcapacity);
4. Vector v=new Vector(Collection c);

Java Program on Vector:

```java
import java.util.Vector;

public class VectorDemo {

	public static void main(String[] args) {

		Vector vector = new Vector();

		System.out.println(vector.capacity());

		for (int i = 1; i <= 10; i++) {

		vector.addElement(i);
		}
		System.out.println(vector.capacity());

		vector.addElement("J");

		System.out.println(vector.capacity());

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

Output on Console:

```
10
10
20
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, J]
```

Java Program on Vector with generics as String:

```java
import java.util.Vector;

public class VectorDemo2 {

	public static void main(String[] args) {


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

		vector.add("sohan");
		vector.add("velocity");
		vector.add("Pune");
		vector.add("Pune");

		while (vector.contains("Pune"))
		{

		vector.remove("Pune");
```

```
                System.out.println("new vector is=" + vector);

                }

                }
}
```

Output on Console:

```
new vector is=[sohan, velocity, Pune]
new vector is=[sohan, velocity]
```

Differences between ArrayList and Vector?

ArrayList	**Vector**
No method is synchronized	Every method is synchronized
At a time multiple Threads are allow to operate on ArrayList object and hence ArrayList object is not Thread safe.	At a time only one Thread is allow to operate on Vector object and hence Vector object is Thread safe.
Relatively performance is high because Threads are not required to wait.	Relatively performance is low because Threads are required to wait
It is non legacy and introduced in 1.2v	It is legacy and introduced in 1.0v

- **Stack:**

  It is the child class of vector.

  It is specially design the class for Last in First Out (LIFO or FILO).

  Constructor

  Stack s= new Stack ();

  Methods

  1. **Object push(Object obj);**
     For inserting an object to stack.

  2. **Object pop();**
     To remove the return top of stack.

  3. **Object peak();**
     To return the top of stack without removal of object.

4. **int Search(Object obj);**
   If specified object is available it returns its offset from top of stack. If object is not available then it return -1.

Java Program on stack:

```
import java.util.Stack;

public class StackList {

        public static void main(String[] args) {

                Stack stack = new Stack();

                stack.push("J");
                stack.push("M");
                stack.push("K");

                System.out.println(stack);

                System.out.println(stack.search("X"));

                // if element not found then return -1

                stack.pop();  //remove first element

                System.out.println(stack);

                stack.peek();

                System.out.println(stack);

                }
}
```

Output on Console:

```
[J, M, K]
-1
[J, M]
[J, M]
```

The 3 cursors of java:

- If we want to get objects one by one from the collection then we should go for cursor. There are 3 types of cursors available in java. They are:
  - Enumeration
  - Iterator
  - List Iterator

Enumeration:

- We can use Enumeration to get objects one by one from the legacy collection objects.
- We can create Enumeration object by using elements () method.

public Enumeration elements();
Enumeration e=v.elements();
using Vector Object

- Enumeration interface defines the following two methods
  1. public boolean hasMoreElements();
  2. public Object nextElement();

Limitations of Enumeration:

- We can apply Enumeration concept only for legacy classes and it is not a universal cursor.
- By using Enumeration we can get only read access and we can't perform remove operations.
- To overcome these limitations sun people introduced Iterator concept in 1.2v.

Java program on enumeration cursor:

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

public class EneumerationDemo {

	public static void main(String[] args) {

	Vector v=new Vector();

	for(int i=0;i<=10;i++){

	v.addElement(i);
	}
```

```
    System.out.println(v);//[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

    Enumeration e=v.elements();

    while(e.hasMoreElements()){

    Integer i=(Integer)e.nextElement();
    if(i%2==0)
    System.out.println(i);   //0 2 4 6 8 10

    }

    System.out.print(v);          //[0, 1, 2, 3, 4, 5, 6, 7, 8,9, 10]

    }

}
```

Output on Console:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
0
2
4
6
8
10
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

♦ Iterator:

- We can use Iterator to get objects one by one from any collection object.
- We can apply Iterator concept for any collection object and it is a universal cursor.
- While iterating the objects by Iterator we can perform both read and remove operations.
- **We can get Iterator object by using iterator () method of Collection interface.**

  public Iterator iterator();

  Iterator itr=c.iterator();
- **Iterator interface defines the following 3 methods.**
  1. public boolean hasNext();
  2. public object next();
  3. public void remove();

- **Limitations of Iterator:**
    - ◊ Both enumeration and Iterator are single direction cursors only. That is we can always move only forward direction and we can't move to the backward direction.
    - ◊ While iterating by Iterator we can perform only read and remove operations and we can't perform replacement and addition of new objects.
    - ◊ To overcome these limitations sun people introduced listIterator concept.

Java program on Iterator cursor:

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

public class IteratorDemo {

	public static void main(String[] args) {

	ArrayList a=new ArrayList();

	for(int i=0;i<=10;i++){

	a.add(i);

	}
	System.out.println(a);//[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

	Iterator itr=a.iterator();
	while(itr.hasNext()){

	Integer i=(Integer)itr.next();
	if(i%2==0)
	System.out.println(i);//0, 2, 4, 6, 8, 10

	else
	itr.remove();

	}
	System.out.println(a);//[0, 2, 4, 6, 8, 10]
	}

}
```

Output on Console:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
0
2
4
6
8
10
[0, 2, 4, 6, 8, 10]
```

- **ListIterator:**
  - ListIterator is the child interface of Iterator.
  - By using listIterator we can move either to the forward direction (or) to the backward direction that is it is a bi-directional cursor.
  - While iterating by listIterator we can perform replacement and addition of new objects in addition to read and remove operations.
  - **By using listIterator method we can create listIterator object.**

     public ListIterator listIterator();

    ListIterator itr=test.listIterator();

    (test is any List object)
  - **ListIterator interface defines the following 9 methods.**
    1. public boolean hasNext();
    2. public Object next(); forward
    3. public int nextIndex();
    4. public boolean hasPrevious();
    5. public Object previous(); backward
    6. public int previousIndex();
    7. public void remove();
    8. public void set(Object new);
    9. public void add(Object new)

Java program on List_Iterator cursor by using hasNext and next method:

```
package collectionInterface;

import java.util.LinkedList;

public class ListIteratorDemo {

        public static void main(String[] args) {

                LinkedList l=new LinkedList();
                l.add("balakrishna");
                l.add("venki");
                l.add("chiru");
                l.add("nag");
                System.out.println(l);//[balakrishna, venki, chiru, nag]
        java.util.ListIterator itr=l.listIterator();
        while(itr.hasNext()) {

        String s=(String)itr.next();

        if(s.equals("venki")){

        itr.remove();
                If}

        }
        System.out.println(l);       //[balakrishna, chiru, nag]
        }

    }
```

Output on Console:

```
[balakrishna, venki, chiru, nag]
[balakrishna, chiru, nag]
```

Java program on ListIterator cursor by using hasPrevious and previous method:

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

public class ListIteratorTest {

    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        for (int i=1 ; i<=5 ; i++) {
              al.add(i);
        }
        ListIterator cursor1 = al.listIterator(al.size());
        while(cursor1.hasPrevious()) {
            int value = (int)cursor1.previous();
            System.out.println(value);
        }
    }
}
```

```
Output :

5
4
3
2
1
```

Java program on ListIterator cursor by using set method :

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

public class ListIteratorTest2 {

    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        for (int i=1 ; i<=5 ; i++) {
            al.add(i);
        }
        ListIterator cursor1 = al.listIterator();

        while(cursor1.hasNext()) {
            int value = (int)cursor1.next();
            if(value == 3) {
                cursor1.set(15);  // to set value
            }
        }
     System.out.println(al);
    }
}
```

```
Output :

[1, 2, 15, 4, 5]
```

Java program on ListIterator cursor by using add method :

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

public class ListIteratorTest2 {

    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        for (int i=1 ; i<=5 ; i++) {
            al.add(i);
        }
        ListIterator cursor1 = al.listIterator();

        while(cursor1.hasNext()) {
            int value = (int)cursor1.next();
            if(value == 3) {
               cursor1.add(44);
            }
        }
     System.out.println(al);
    }
}
```

```
Output :

[1, 2, 3, 44, 4, 5]
```

Java program on ListIterator cursor to get Next Index and Previous Index :

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

public class ListIteratorTest2 {

	public static void main(String[] args) {
		ArrayList al = new ArrayList();
		for (int i=1 ; i<=5 ; i++) {
			al.add(i);
		}
		ListIterator cursor1 = al.listIterator();

		while(cursor1.hasNext()) {
			int value = (int)cursor1.next();
			if(value == 3) {
			   int Nextindex = cursor1.nextIndex();  // to get nextindex
			   System.out.println("Next Index is : " + Nextindex);
			   int previousindex = cursor1.previousIndex();  // to get
previousindex
			   System.out.println("Next Index is : " + previousindex);
			}
		}
	  System.out.println(al);
	}
}
```

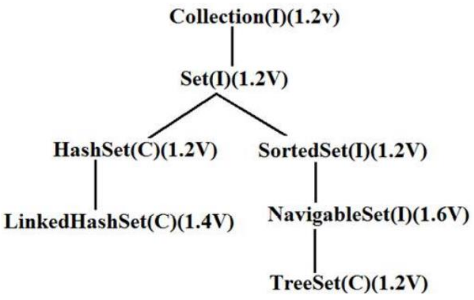
```
Output :


Next Index is : 3
Next Index is : 2
[1, 2, 3, 4, 5]
```

Comparison of Enumeration, Iterator and ListIterator?

Property	Enumeration	Iterator	ListIterator
Is it legacy?	Yes	No	No
It is applicable for?	Only legacy classes.	Applicable for any collection object.	Applicable for only list objects.
Movement?	Single direction cursor(forward)	Single direction cursor(forward)	Bi-directional
How to get it?	By using elements() method.	By using iterator()method.	By using listIterator() method.
Accessibility?	Only read.	Both read and remove.	Read/remove/replace/add.
Methods	hasMoreElement() nextElement()	hasNext() next() remove()	9 methods.

▪ Set Interface:

- It is the child interface of collection.
- If we want to represent a group of individual objects as a single entity **where duplicates are not allow and insertion order is not preserved** then we should go for set interface.
- **Set interface does not contain any new method we have to use only collection interface methods.**



♦ HashSet:

- The underlying data structure is Hashtable.
- Insertion order is not preserved and it is based on hash code of the objects.
- Duplicate objects are not allowed.
- If we are trying to insert duplicate objects we won't get compile time error and runtime error add() method simply returns false.
- Heterogeneous objects are allowed.
- Null insertion is possible.(only once)
- Implements Serializable and Cloneable interfaces but not Random Access.
- HashSet is best suitable, if our frequent operation is "Search".
- **Constructors:**

    1. HashSet h=new HashSet ();

    Creates an empty HashSet object with default initial capacity 16 and default fill ratio 0.75(fill ratio is also known as load factor).

    2. HashSet h=new HashSet (int initialcapacity);

    Creates an empty HashSet object with the specified initial capacity and default fill ratio 0.75.

    3. HashSet h=new HashSet (int initialcapacity, float fill ratio);

    4. HashSet h=new HashSet (Collection c);

- **Note : After filling how much ratio new HashSet object will be created, the ratio is called "FillRatio" or "LoadFactor".**

Java Program on HashSet:

```
import java.util.HashSet;

public class HashSetTest {

    public static void main(String[] args) {
        HashSet test = new HashSet();
        for ( char ch= 'A' ; ch < 'H' ; ch++) {
            test.add(ch);
        }
        System.out.println(test);
        test.add(null);    //add single only once;
        test.add('B');     //trying to add duplicate but return false
        System.out.println(test);
    }
}
```

Output on Console:

```
[A, B, C, D, E, F, G]
[null, A, B, C, D, E, F, G]
```

♦ LinkedHashSet:

- It is the child class of HashSet.
- Introduced in 1.2 version.
- It is exactly same as HashSet but except the following difference.

HashSet	**LinkedHashSet**
The underlying data structure is Hashtable.	The underlying data structure is Hashtable + LinkedList (that is hybrid data structure).
Insertion order is not preserved.	Insertion order is preserved.
Introduced in 1.2 version.	Introduced in 1.4 version.

Java Program on Link_Hash_Set:

```java
import java.util.LinkedHashSet;

public class LinkedHashSetTest {

        public static void main(String[] args) {
          LinkedHashSet  test = new LinkedHashSet();
          test.add("B");
          test.add("C");
          test.add("D");
          test.add("Z");
          test.add(null);
          test.add(10);
          System.out.println(test.add("Z"));//false
          System.out.println(test);//[B, C, D, Z, null, 10]
        }
}
```

Output on Console:

```
false
[B, C, D, Z, null, 10]
```

Note: LinkedHashSet and LinkedHashMap commonly used for implementing "cache applications" where insertion order must be preserved and duplicates are not allowed.

SortedSet:

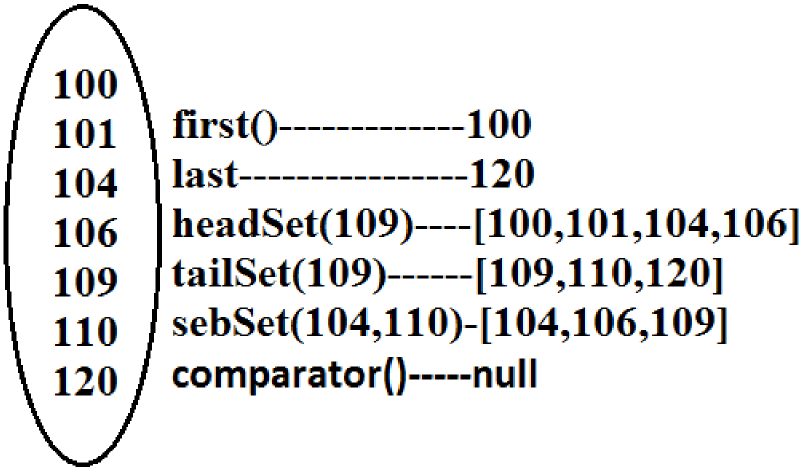
- It is child interface of Set
- If we want to represent a group of "**unique objects" where duplicates are not allowed** and all objects must be inserting according to some sorting order then we should go for SortedSet interface.
- That sorting order can be either default natural sorting (or) customized sorting order.
- **SortedSet interface define the following 6 specific methods.**
  1. **Object first();**
  2. **Object last();**
  3. **SortedSet headSet(Object obj);**

     Returns the SortedSet whose elements are <obj.
  4. **SortedSet tailSet(Object obj);**

     It returns the SortedSet whose elements are >=obj.
  5. **SortedSet subset(Object o1,Object o2);**

     Returns the SortedSet whose elements are >=o1 but <o2.
  6. **Comparator comparator();**

     Returns the Comparator object that describes underlying sorting technique.
     If we are following default natural sorting order then this method returns null.


100
101
104
106
109
110
120
first()-------------100
last----------------120
headSet(109)----[100,101,104,106]
tailSet(109)------[109,110,120]
sebSet(104,110)-[104,106,109]
comparator()-----null

TreeSet:

- Underlying data structure is balanced tree.
- Duplicates objects are not allowed.
- Insertion order is not preserved.
- All the objects will be inserted according to some sorting order.
- Heterogeneous objects are not allowed.

- If we are trying to insert the heterogeneous objects then will get run time exception saying classcastexception.
- Null insertion is possible (only once).
- **Constructor**
    1. **TreeSet t = new TreeSet();**

        Create the empty TreeSet object where elements will be inserted according to default natural sorting order.

    2. **TreeSet t = new TreeSet(Comparator c);**

        Creates an empty TreeSet object where all objects will be inserted according to customized sorting order specified by Comparator object.

    3. **TreeSet t= new TreeSet(SortedSet s);**
    4. **TreeSet t= new TreeSet(Collection c);**

- **Null Acceptance**
    1. For the empty TreeSet as the 1st element "null" insertion is possible but after Inserting that null if we are trying to insert any other we will get NullPointerException.
    2. **For the non-empty TreeSet if we are trying to insert null then we will get NullPointerException**

Java Program on TreeSet:

```
import java.util.TreeSet;

public class TreeSetDemo {

	public static void main(String[] args) {

		TreeSet treeSet = new TreeSet();

		treeSet.add("Jay");
		treeSet.add("ram");
		treeSet.add("Shyam");

		System.out.println(treeSet);   //dictionary insertion order

		}

}
```

Output on Console:

```
[Jay, Shyam, ram]
```

Comparison between Set Implemented classes-

Property	**HashSet**	**LinkedHashSet**	**TreeSet**
Underlying data structure	Hashtable	Hashtable+ LinkedList	Balanced Tree
Insertion order	Not preserved	Preserved	Not Applicable
Sorting order	Not applicable	Not applicable	Applicable
Heterogeneous objects	allowed	allowed	Not allowed
Duplicates objects	Not allowed	Not allowed	Not allowed
Null acceptance	Allowed(only once)	Allowed(only once)	We will get nullpointer exception.

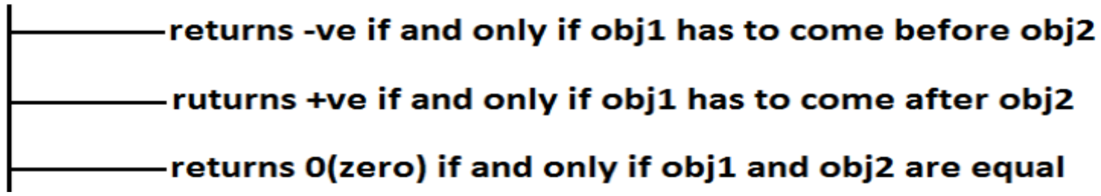
Comparable interface:

- Comparable interface present in java.lang package and contains only one method.

compareTo() method.

public int compareTo(Object obj);

Example: obj1.compareTo(obj2);


returns -ve if and only if obj1 has to come before obj2

ruturns +ve if and only if obj1 has to come after obj2

returns 0(zero) if and only if obj1 and obj2 are equal

Java Program on comparable interface:

```
public class ComparableInterfaceTest {
	public static void main(String[] args) {
		System.out.println("C".compareTo("A"));  //Output 2
		System.out.println("A".compareTo("Z"));  //Output -25
		System.out.println("B".compareTo("B"));  //Output 0
		System.out.println("T".compareTo("S"));   //Output 1
		System.out.println("T".compareTo("U"));   //Output -1
	}
}
```

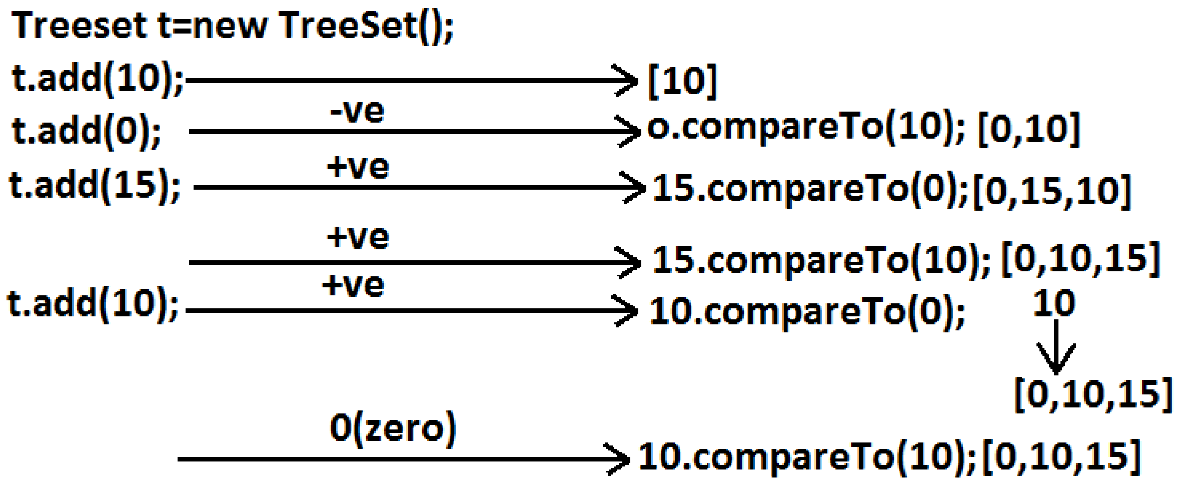
Output on Console:

```
2
-25
0
1
-1
```

If we are depending on default natural sorting order then internally JVM will use **compareTo ()** method to arrange objects in sorting order.

```
import java.util.TreeSet;

public class CompareToTest {

        public static void main(String[] args){
                TreeSet t=new TreeSet();
                t.add(10);
                t.add(0);
                t.add(15);
                t.add(10);
                System.out.println(t); //[0, 10, 15]
        }
}
```

CompareTo ( ) method analysis:

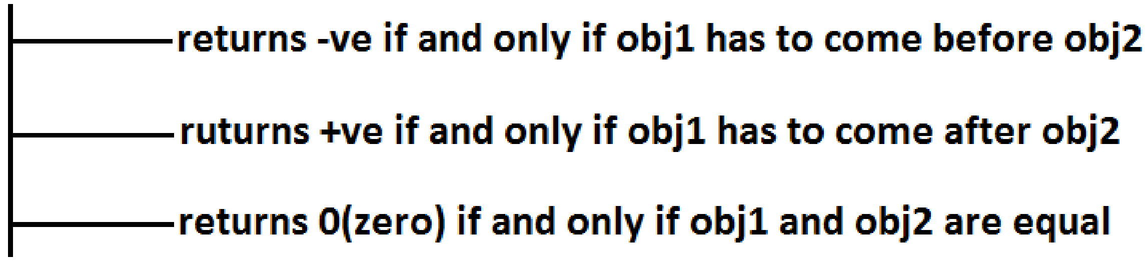


- If we are not satisfying with default natural sorting order (or) if default natural Sorting order is not available then we can define our own customized sorting by Comparator object.
- **Comparable meant for default natural sorting order.**
- **Comparator meant for customized sorting order.**

Comparator interface:

- Comparator interface present in java.util package this interface defines the following 2 methods.

    1. **public int compare(Object obj1,Object Obj2);**



returns -ve if and only if obj1 has to come before obj2

ruturns +ve if and only if obj1 has to come after obj2

returns 0(zero) if and only if obj1 and obj2 are equal

2. **public boolean equals(Object obj);**
    - Whenever we are implementing Comparator interface we have to provide implementation only for **compare ()** method.
    - Implementing **equals ()** method is optional because it is already available from Object class through inheritance.

Note: If we are depending on default natural sorting order then the objects should be “homogeneous and comparable" otherwise we will get ClassCastException. If we are defining our own sorting by Comparator then objects "need not be homogeneous and comparable".

Requirement: Write a program to insert integer objects into the TreeSet where the sorting order is descending order

```
import java.util.TreeSet;
public class TreeSetDemo {
	public static void main(String[] args) {
		TreeSet test = new TreeSet(new MyComparator());
		 test.add(18);
		 test.add(10);
		 test.add(5);
		 test.add(20);
		 System.out.println(test);
	}
}
}
```

Output on Console:

```
[20, 18, 10, 5]
```

```
import java.util.Comparator;
public class MyComparator implements Comparator<Integer>{

	@Override
	public int compare(Integer o1, Integer o2) {
		return -o1.compareTo(o2);  // to sort object in decending order
		//return o1.compareTo(o2);  // to sort object in Ascending order
	}
}
```

Difference between Comparable and Comparator?

Comparable	**Comparator**
Comparable meant for default natural sorting order	Comparator meant for customized sorting order
Present in java.lang package.	Present in java.util package
Contains only one method. **compareTo()** method	Contains 2 methods. **Compare() method**. **Equals() method**
String class and all wrapper Classes implements Comparable interface.	The only implemented classes of Comparator are Collator and RuleBasedCollator. (used in GUI)

Comparable vs Comparator:

1. **For predefined Comparable classes default natural sorting order is already available if we are not satisfied with default natural sorting order then we can define our own customized sorting order by Comparator.**

2. **For predefined non Comparable classes [like StringBuffer] default natural sorting order is not available we can define our own sorting order by using Comparator object.**

3. **For our own classes [like Customer, Student, and Employee] we can define default natural sorting order by using Comparable interface. The person who is using our class, if he is not satisfied with default natural sorting order then he can define his own sorting order by using Comparator object.**

What is Synchronization?

- It is a process by which we control the accessibility of multiple threads to a particular shared resource.

Problem Without synchronization?

- Final outcome is not deterministic.
- Thread interference.

Advantages of synchronization?

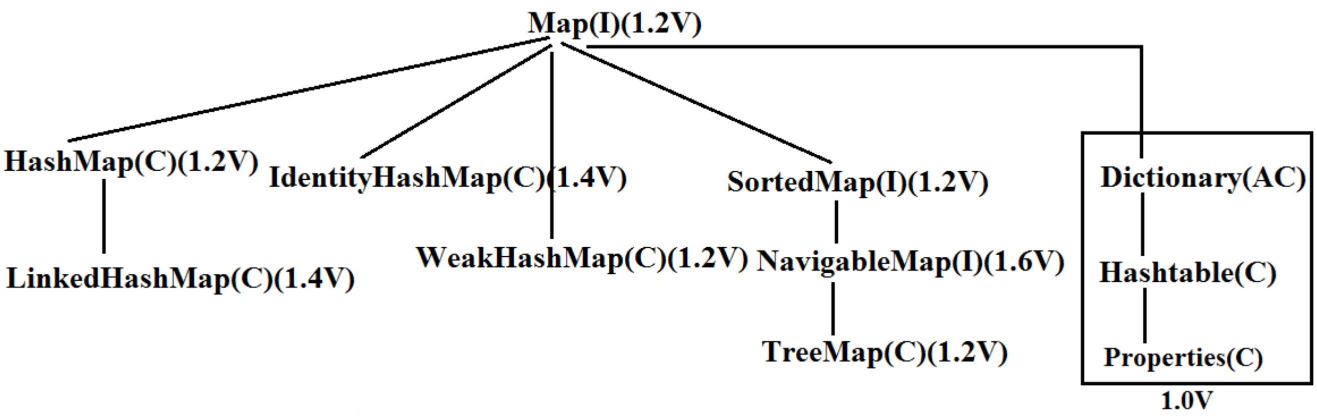
- Final outcome is deterministic.
- No Thread interference.

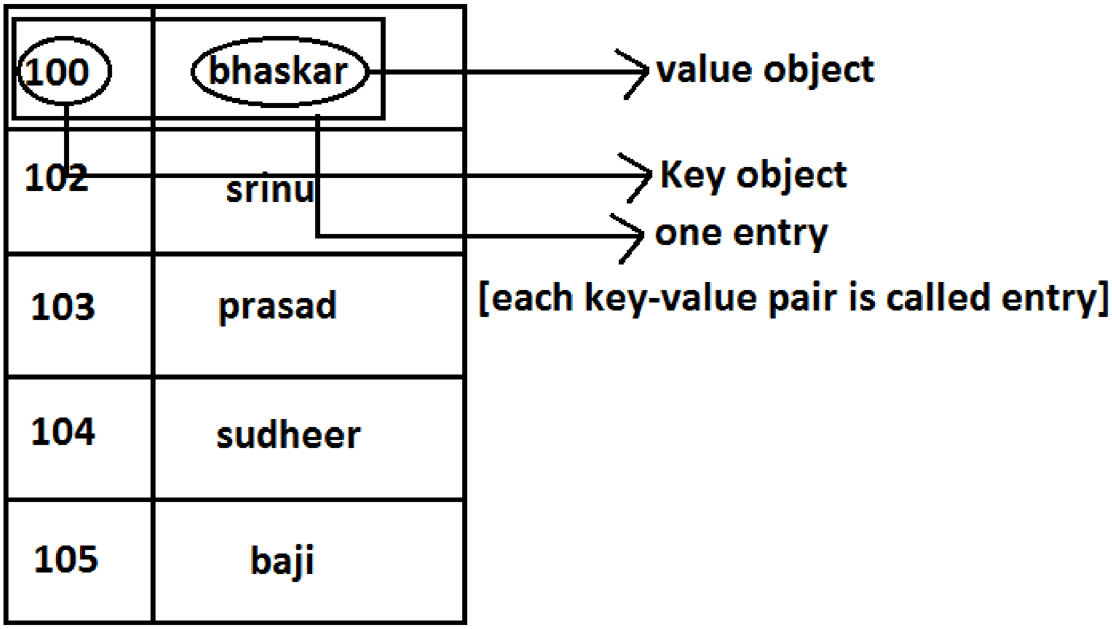
Disadvantage of synchronization?

- Increase the waiting time period of thread.
- Create performance issue.

Map Interface:

- If we want to represent a group of objects as **"key-value"** pair then we should go for Map interface.
- Both key and value are objects only.
- Duplicate keys are not allowed but values can be duplicated.
- Each key-value pair is called **"one entry".**
- **Map interface is not child interface of Collection and hence we can't apply Collection interface methods here.**





100	bhaskar
102	srinu
103	prasad
104	sudheer
105	baji

[each key-value pair is called entry]

- **Map interface defines the following specific methods**

1. **Object put(Object key, Object value);**

   To add an entry to the Map, if key is already available then the old value replaced with new value and old value will be returned.

2. **void putAll(Map m);**
3. **Object get(Object key);**
4. **Object remove(Object key);**
   It removes the entry associated with specified key and returns the corresponding value.
5. **boolean containsKey(Object key);**
6. **boolean containsValue(Object value);**
7. **boolean isEmpty();**
8. **Int size();**
9. **void clear();**
10. **Set keySet();**
    The set of keys we are getting.
11. **Collection values();**
    The set of values we are getting.
12. **Set entrySet();**
    The set of entrySet we are getting.

- **Entry Interface:**
  - Each key-value pair is called one entry. Hence Map is considered as a group of entry Objects, without existing Map object there is no chance of existing entry object hence interface entry is define inside Map interface (inner interface).

HashMap:

- A HashMap is class which implements the Map interface
- The underlying data structure is Hashtable.
- Duplicate keys are not allowed but values can be duplicated.
- **Insertion order is not preserved and it is based on hash code of the keys.**
- Heterogeneous objects are allowed for both key and value.
- Null is allowed for keys (only once) and for values (any number of times).
- It stores values based on key.
- It has 16 size and internally it will increase the size by double, so new size will be 32, 64,128.
- It is unordered, which means that the key must be unique
- It may have null key-null value
- For adding elements in HashMap we use the put method.
- **It is best suitable for Search operations.**
- Return type of put method is Object.

- **Constructors:**

1. **HashMap m=new HashMap();**
   Creates an empty HashMap object with default initial capacity 16 and default fill ratio "0.75".
2. **HashMap m=new HashMap(int initialcapacity);**
3. **HashMap m =new HashMap(int initialcapacity, float fillratio);**
4. **HashMap m=new HashMap(Map m);**

Differences between HashMap and Hashtable?

HashMap	**Hashtable**
No method is synchronized.	Every method is synchronized.
Multiple Threads can operate simultaneously on HashMap object and hence it is not Thread safe.	Multiple Threads can't operate simultaneously on Hashtable object and hence Hashtable object is Thread safe.
Relatively performance is high.	Relatively performance is low.
Null is allowed for both key and value.	Null is not allowed for both key and value otherwise we will get NullPointerException
It is non legacy and introduced in 1.2v	It is legacy and introduced in 1.0v

How to get synchronized version of HashMap:

- By default HashMap object is not synchronized. But we can get synchronized version by using the following method of Collections class.
- public static Map synchronizedMap(Map m1)

- **Java Program on HashMap:**

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

public class HashMapDemo {

        public static void main(String[] args){

                HashMap m=new HashMap();
                m.put("chiranjeevi",700);
                m.put("balaiah",800);
                m.put("venkatesh",200);
                m.put("nagarjuna",500);

                System.out.println(m);//{balaiah=800, chiranjeevi=700, venkatesh=200, 
nagarjuna=500}

                System.out.println(m.put("chiranjeevi",100));//700

                Set s=m.keySet();

                System.out.println(s);    //  [balaiah, chiranjeevi, venkatesh, 
nagarjuna]

                Collection c=m.values();
                System.out.println(c);//[800, 100, 200, 500]
                Set s1=m.entrySet();

                System.out.println(s1);//[balaiah=800, chiranjeevi=100, venkatesh=200, 
nagarjuna=500]

                Iterator itr=s1.iterator();

                while(itr.hasNext()) {
                        Map.Entry m1=(Map.Entry)itr.next();
                        System.out.println(m1.getKey()+"......"+m1.getValue());
                        if(m1.getKey().equals("nagarjuna")){
                        m1.setValue(1000);
                        }
                }
                System.out.println(m);  //{balaiah=800, chiranjeevi=100, venkatesh=200, 
nagarjuna=1000}
```

```
        }
}
```

Output on Console:

```
{balaiah=800, chiranjeevi=700, venkatesh=200, nagarjuna=500}
700
[balaiah, chiranjeevi, venkatesh, nagarjuna]
[800, 100, 200, 500]
[balaiah=800, chiranjeevi=100, venkatesh=200, nagarjuna=500]
balaiah......800
chiranjeevi......100
venkatesh......200
nagarjuna......500
{balaiah=800, chiranjeevi=100, venkatesh=200, nagarjuna=1000}
```

LinkedHashMap-

- A LinkedHashMap is a ‘hashtable and linked list implementation of the map interface with a predictable iteration order.
- It is the same as HashMap except it maintains an insertion order i.e. ordered

HashMap	**LinkedHashMap**
The underlying data structure is Hashtable.	The underlying data structure is a combination of Hashtable+ LinkedList.
Insertion order is not preserved.	Insertion order is preserved.
It is introduced in 1.2v	It is introduced in 1.4v.

Java Program on LinkedHashMap:

```
import java.util.LinkedHashMap;

public class HashMapDemo2 {

        public static void main(String[] args) {

                LinkedHashMap linkedHashMap = new LinkedHashMap();
                linkedHashMap.put(10, "ajay");
                linkedHashMap.put(11, "ram");
                linkedHashMap.put(12, "shyam");
                System.out.println(linkedHashMap);
        }

}
```

Output on Console:

```
{10=ajay, 11=ram, 12=shyam}
```

- ## IdentityHashMap-
    - It is exactly same as HashMap except the following differences:
        1. In the case of HashMap JVM will always use ".equals ()"method to identify duplicate keys, which is meant for content comparision.
        2. But in the case of IdentityHashMap JVM will use== (double equal operator) to identify duplicate keys, which is meant for reference comparision.

    - ### Java Program on LinkedHashMap:

```
        import java.util.HashMap;
    public class HashMapTest {
        public static void main(String[] args) {
            HashMap test = new HashMap();
            Integer i1 = new ~~Integer~~(10);
            Integer i2 = new ~~Integer~~(10);
            test.put(i1, "Kalyan");
            test.put(i2, "pavan");
            System.out.println(test);
            System.out.println(test.get(10));
        }
    }
```

Output on Console:

```
{10=pavan}
pavan
```

- In the above program i1 and i2 are duplicate keys because i1.equals(i2) returns true
- In the above program if we replace HashMap with IdentityHashMap then i1 and i2 are not duplicate keys because i1==i2 is false hence in this case the output is {10=pavan, 10=kalyan}.

- ## WeakHashMap-
    - It is exactly same as HashMap except the following differences.
        1. In the case of normal HashMap, an object is not eligible for GC even though it doesn't have any references if it is associated with HashMap. That is HashMap dominates garbage collector.
        2. But in the case of WeakHashMap if an object does not have any references then it's always eligible for GC even though it is associated with WeakHashMap that is garbage collector dominates WeakHashMap.

- ### Java Program on WeakHashMap:

```
import java.util.WeakHashMap;
public class WeakHashMapDemo {
   public static void main(String[] args) throws InterruptedException {
           WeakHashMap m = new WeakHashMap();
           Temp t = new Temp();
           m.put(t, "Ashok");
           System.out.println(m);//{Temp=ashok}
           t=null;
           System.gc();
           Thread.sleep(5000);
           System.out.println(m);//{}
   }
}


   public class Temp {

          public String toString ( ) {
                return "Temp";
          }

          public void finalize(){
          System.out.println("finalize() method called");
          }
   }
```

Output on Console:

```
{Temp=Ashok}
finalize() method called
{}
```

SortedHashMap-

- It is the child interface of Map.
- If we want to represent a group of key-value pairs according to some sorting order of keys then we should go for SortedMap.
- Sorting is possible only based on the keys but not based on values**.**
- **SortedMap interface defines the following 6 specific methods**
    1. **Object firsyKey();**
    2. **Object lastKey();**
    3. **SortedMap headMap(Object key);**
    4. **SortedMap tailMap(Object key);**
    5. **SortedMap subMap(Object key1,Object key2);**
    6. **Comparator comparator();**

TreeMap-

- The underlying data structure is RED-BLACK Tree.
- Duplicate keys are not allowed but values can be duplicated.
- Insertion order is not preserved and all entries will be inserted according to some sorting order of keys.
- If we are depending on default natural sorting order keys should be homogeneous and Comparable otherwise we will get ClassCastException.
- If we are defining our own sorting order by Comparator then keys can be heterogeneous and non-Comparable.
- There are no restrictions on values they can be heterogeneous and non-Comparable.
- For the empty TreeMap as first entry null key is allowed but after inserting that entry if we are trying to insert any other entry we will get NullPointerException.
- For the non-empty TreeMap if we are trying to insert an entry with null key we will get NullPointerException.
- There are no restrictions for null values.
- **Constructors:**
  1. **TreeMap t=new TreeMap();**
     For default natural sorting order.
  2. **TreeMap t=new TreeMap(Comparator c);**
     For customized sorting order.
  3. **TreeMap t=new TreeMap(SortedMap m);**
  4. **TreeMap t=new TreeMap(Map m);**

```
import java.util.TreeMap;
public class TreeMapDemo {
      public static void main(String[] args) {
            TreeMap t = new TreeMap ();
            t.put(100, "ZZZ");
            t.put(200, "YYY");
            t.put(300, "XXX");
            t.put(104, 106);
            t.put(107, null);
//          t.put("FFF", "XXX"); // ClassCastException
//          t.put(null, "XXX");  //NullPointerException
            System.out.println(t);
      }
}
```

Output on Console:

```
{100=ZZZ, 104=106, 107=null, 200=YYY, 300=XXX}
```

```
import java.util.TreeMap;
public class TreeMapTest {
        public static void main(String[] args) {
                TreeMap t = new TreeMap(new MyComparator());
                t.put("XXX",10);
                t.put("AAA",20);
                t.put("ZZZ",30);
                t.put("LLL",40);
                System.out.println(t); // {ZZZ=30, XXX=10, LLL=40, AAA=20}
        }
}

import java.util.Comparator;
public class MyComparator implements Comparator{
        public int compare(Object obj1 ,Object obj2){
                String s1=obj1.toString();
                String s2=obj2.toString();
                return s2.compareTo(s1);
        }
}
```

- Hashtable-

  - The underlying data structure is Hashtable.
  - Insertion order is not preserved and it is based on hash code of the keys.
  - Heterogeneous objects are allowed for both keys and values.
  - Null key (or) null value is not allowed otherwise we will get NullPointerException.
  - Duplicate keys are allowed but values can be duplicated.
  - Every method present inside Hashtable is synchronized and hence Hashtable objet is Thread-safe.
  - **Constructors:**
    1. **Hashtable h=new Hashtable();**
       Creates an empty Hashtable object with default initialcapacity 11 and default fill ratio 0.75.
    2. **Hashtable h=new Hashtable(int initialcapacity);**
    3. **Hashtable h=new Hashtable(int initialcapacity, float fillratio);**
    4. **Hashtable h=new Hashtable (Map m);**

```
package com.test;
import java.util.Hashtable;
public class HashTableDemo {
      public static void main(String[] args) {
            Hashtable hashTable = new Hashtable();
            hashTable.put(10, "ram");
            hashTable.put(11, "sohan");
            System.out.println(hashTable);
      }
}
```

Output on Console:

```
{10=ram, 11=sohan}
```

Comparison between HashMap, LinkedHashMap, TreeMap and HashTable:

Topic	HashMap	LinkedHashMap	TreeMap	HashTable
Duplicate Key	Not Allowed	Not Allowed	Not Allowed	Not Allowed
Ordering	Unordered	Maintains insertion order	Maintains in Accessing order	Unordered
Null (Key Value)	Allow	Allow	key Not allowed but value is Iterator	Not Allowed
Accessing Elements	Iterator	Iterator	Iterator	Iterator
Thread Safety	No	No	No	Yes

```
import java.util.Hashtable;
public class HashtableDemo {
      public static void main(String[] args) {
            Hashtable h=new Hashtable();
            h.put(new Temp2(5),"A");
            h.put(new Temp2(2),"B");
            h.put(new Temp2(6),"C");
            h.put(new Temp2(15),"D");
            h.put(new Temp2(23),"E");
            h.put(new Temp2(16),"F");
            System.out.println(h);  //{6=C, 16=F, 5=A, 15=D, 2=B, 23=E}
      }
}


public class Temp2 {

      int i;
      Temp2(int i){
            this.i=i;
      }

      public int hashCode(){
            return i;
      }
```

```
    public String toString(){
        return i+"";
    }
}
```

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

public class HashMapDemo4 {

    public static void main(String[] args) {

        HashMap<Integer, String> map = new HashMap<Integer, String>();
        map.put(10, "Ram");
        map.put(20, "yogesh");
        map.put(30, "sohan");

        Set<Integer> s = map.keySet(); // s contain all the keys only.

        for (int i : s) {
            System.out.println("Key==" + i);
            System.out.println("value=" + map.get(i));
        }
    }
}
```

Output on Console:

```
Key==20
value=yogesh
Key==10
value=Ram
Key==30
value=sohan
```

Exception Handling:

- ## Exception:

  - An Exception is an unwanted or unexpected condition which disturbs our normal flow of execution.
  - Once Exception occurred remaining part of program will not be executed.
  - So, it is our responsibility to handle the exception.
  - Exception handling doesn't means, we are resolving an exception it is just like providing an alternate solution so that even though exception happens our program should work properly.

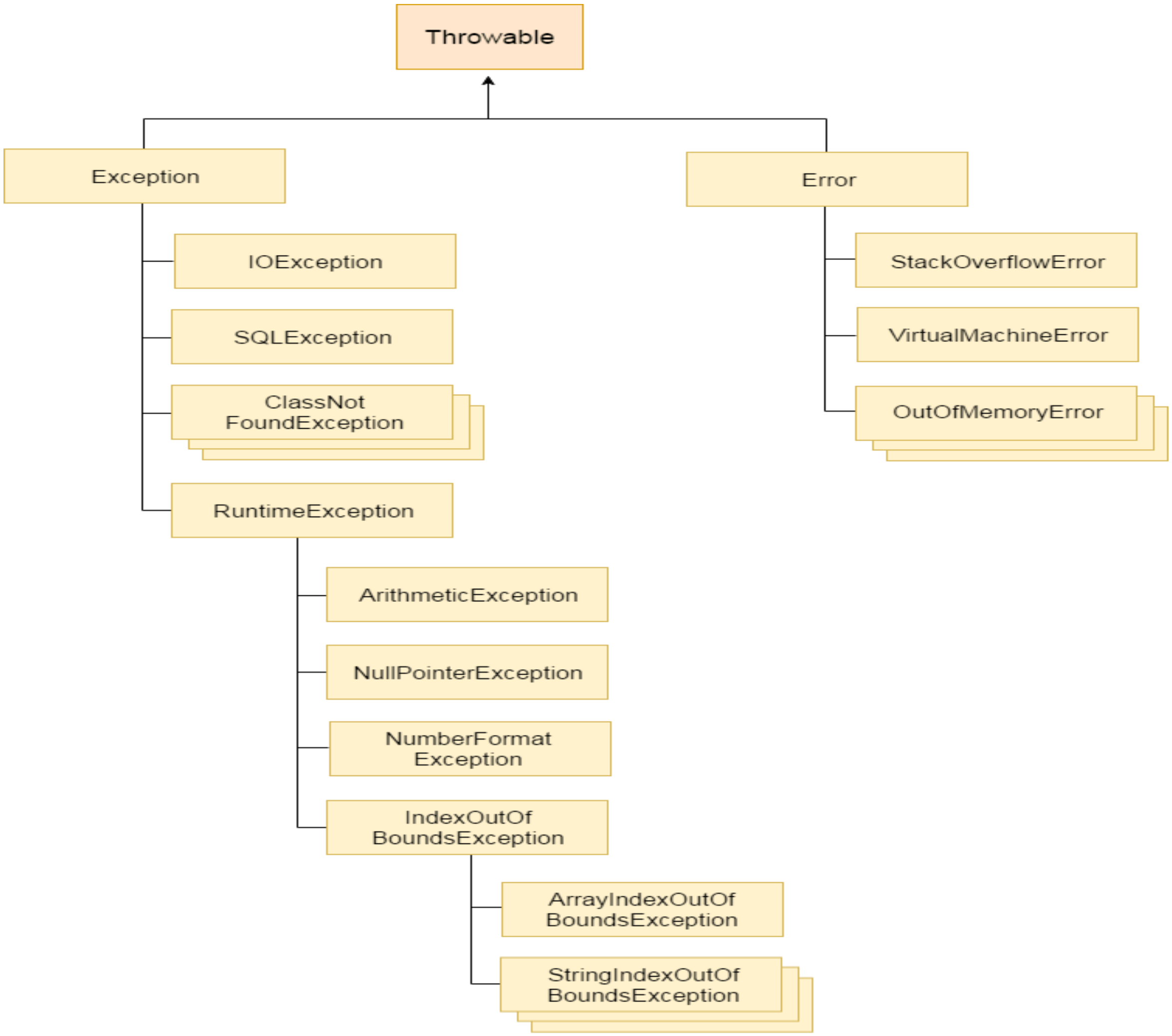
- **Exception handling:**
    - **Definition: -** Exception handling is nothing but to handle the abnormal termination of a program into normal termination. And make the program to execute completely even though there is an exception caused during the execution.

- **Exception Hierarchy:**
    - Object class is a super class to all predefine and user define classes of java.
    - Throwable class is a super class to "Exception" class and "Error" class.
    - Exception class is a super class to RuntimeException class and other Exception classes.
    - All the Exception classes belongs to java.lang package.

- **Depending on Hierarchy, Exceptions are divided into 2 types**
    - Checked Exception (Compile time Exceptions)
    - Unchecked Exception (Run time Exceptions)

- **Checked Exception**
    - Exception which are checked (identified or found out) during compile time by compiler, such type of exception are called as Checked Exceptions.
    - Exception classes which are directly inheriting Exception class except RuntimeExceptionclass is called as checked exception.
    - Checked Exceptions are also called as Compile time Exception.
    - Examples (Classes) of Checked Exceptions are: **InterruptedException ClassNotFoundException, -SQLException, and FileNotFoundException etc.**

- **Unchecked Exception**
    - Exception which are checked (identified or found out) during Runtime or execution time, such type of exception are called as Unchecked Exceptions.
    - In case of Unchecked Exception our program will at least compiles successfully.
    - Unchecked Exceptions are also called as Runtime Exceptions.
    - RuntimeExceptionclass is a super class to all Unchecked Exception classes.
    - **Examples(Classes) of Unchecked Exceptions are :-**
        1. ArithmeticException
        2. ArrayIndexOutOfBoundsException
        3. NullPointerException
        4. StringIndexOutofBoundsException
        5. ClassCastException
        6. NumberFormatException



- **Error :**

- An Error is an irrecoverable Condition i.e, if error occured it is not under programmers control to get over it.
- For Ex: if we develop any program whose size is 4gb but our system's storage is 3gb so such condition is not in programmers control and such situation is referred as Error.
- **Examples(Classes) of Error are :**

  1. StackoverFlowError
  2. VirtualMemoryError
  3. 404pagenotfound

Differences between Error and Exception?

Error	**Exception**
An error is caused due to lack of system resources.	An exception is caused because of some problem in code.
An error is irrecoverable i.e, an error is a critical condition cannot be handled by code of program.	An Exception is recoverable i.e, we can have some alternate code to handle exception.
There is no ways to handle error.	We can handle exception by means of try and catch block.
As error is detected program is terminated abnormally.	As Exception is occurred it can be thrown and caught by catch block.
There is no classification for Error.	Exceptions are classified as checked and unchecked.
Errors are define in java.lang. Error package	Exceptions are define in java.lang. Exception package.

Valid Combinations -

- try{} catch{} finally{}
- try{} finally{}
- try{} catch{} catch{} finally{}
- try{} catch{} finally{} try{} catch{} finally{}

Invalid Combinations

- try{} catch{} finally{} finally{}
- catch{} finally{}
- finally{} try{} catch{}

There are two ways to handle the exception:

- try-catch-finally
- throws keyword

- **try-catch- finally:**
    - Inside try block we generally writes the risky code which can cause an exception
    - In the catch block we writes the code which can tell us to bypass the situation on which we got an exception. Only that particular catch will get execute which has written for that particular exception.
    - For example if in try block we gets Arithmetic exception then there should be a catch block with Arithmetic exception otherwise program will get terminate abnormally.
    - Finally block executes every time whether we gets an exception or not. It is basically to perform cleanup activities.

Java Program On try-catch-finally based exception Handling:

```
public class Test {

	public static void main(String[] args) {

		int i =10;
		int j=0;

		System.out.println("Before arrival of exception");

		try {
			int k=i/j;
			System.out.println(k);
		}
		catch (Exception e) {
			System.out.println("exception handled in catch block" +
e.getMessage());
		}

		finally {
			System.out.println("Finally block is running");
		}

		System.out.println("After handling of exception");
	}
}
```

Output on Console:

```
Before arrival of exception
exception handled in catch block/ by zero
Finally block is running
After handling of exception
```

- If the type of exception which is inside the try block is covered by catch block then the exception will get **handle and the program gets terminate in a normal way.**
- If the type of exception inside try block **is not been covered by any of the catch blocks then the program would get terminate abnormally.**

♦ throws keyword

- By using throws keyword we can handle the compile time error for exception handling but if there is an exception caused during runtime then it cannot protect from abnormal termination of program.
- It is recommended to use throws keyword for checked exception.

Java Program On throws keyword based exception Handling:

```
public class Test {

	public static void main(String[] args) throws InterruptedException {
		int i=10;
		int j=0;
		Thread.sleep(2000);
		System.out.println("Exception handled for thread by Throws");
	}
}
```

Output on Console:

```
Exception handled for thread by Throws.
```

♦ Throw Keyword:

- By using throw keyword we can throw the exception at a particular situation in the program.
- It is generally used for throwing **customize exception (Exceptions which are defined by user).**

Java Program On throw keyword based exception Handling:

```
public class ThrowKeyword {

	public static void main(String[] args) {
		int i=8;
		int j=20;

		if (i>5)  {
			throw new ArithmeticException ("Exception occur at particular 
point ");
		}

		if (j>20)  {
```

```
                        System.out.println("No any exception");
                }
                System.out.println("normal run of the program");
        }
}
```

Output on Console:

```
Exception in thread "main" java.lang.ArithmeticException: Exception occur at
particular point
        at collectionPractice.ThrowKeyword.main(ThrowKeyword.java:10)
```

Differences between throw and throws?

throw	**throws**
throw keyword is used to create Exception object explicitly	throws is used to declare the Exception
throw keyword is used inside the method	throws keyword is used with method declaration
Syntax: throw new ExceptionName(Excp description); Ex: throw new ArithmeticException("MyExcept");	Ex: method declaration Exceptionname public void fly() throws InterruptedException
throw keyword is mainly used for Userdefine exception	throws keyword is mainly used for checked exception
Using throw keyword, we can throw only one exception at a time.	one exception at a time Using throws keyword, we can declare multiple exceptions at a time.
throw new MinBalException("Zero");	public void check() throws InterruptedException,SQLException

Basically, "finally" is used to keep an important code which should not be skipped at any condition like closing of data base connection or closing of opened file etc.

Difference between final, finally and finalize?

- **Final :**
  - Final is used to apply restrictions on class, method and variables.
  - Final class can’t be inherited.
  - Final method can’t be overridden.
  - Final variable cannot be changed.
  - Final is keyword.

- **Finally:**
  - Finally is used to place important code, it will be executed whether the exception is handled or not.
  - **Basically, "finally" is used to keep an important code which should not be skipped at any condition like closing of data base connection or closing of opened file etc.**
  - Finally is a block.

- **Finalize:**
  - Finalize is used to perform clean up activity just before object garbage collected.
  - Finalize is a method.
  - Finalize ( ) is a method which generally called by garbage collector or JVM to cut off the remaining connections of the unused object and destroy the same for memory optimization.

- **Finally and Close() :**
  - close () statement is used to close all the open streams in a program.
  - It’s a good practice to use close () inside finally block.
  - Since finally block executes even if exception occurs so you can be sure that all input and output streams are closed properly regardless of whether the exception occurs or not.

- **Finally block and System.exit()**
  - System.exit(0) gets called without any exception then finally won’t execute. However if any exception occurs while calling System.exit (0) then finally block will be executed.

Can we write only try block without catch block?

- No, a try block should always followed by either catch or finally block.

Can we write any statement between try and catch block?

- No, we cannot write any statement between try and catch block. Immediately after try block there should be catch or finally block.

If we don't know exception type, what type should we mention in catch block?

- When we do not know Exception type, we can mention it as ExceptionClass type or Throwable type.
- Because, Exception is a super class to all the class and during up casting we studied that superclass can hold reference of subclass object.
    1. Ex: Exception e=new ArithmeticException()
    2. Ex: Throwable e1=new ArithmeticException ()