Array-
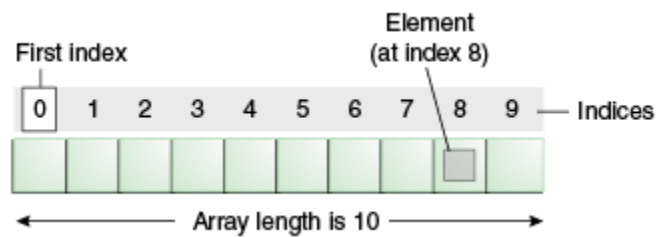
- Array is a collection of similar type of elements which has contiguous memory location.
- Java array is an object which contains elements of a similar data type.
- We can store only a fixed set of elements in a Java array.
- Array in Java is index-based, the first element of the array is stored at 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.**

```java
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++) {
```

```java
                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

```java
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:**

```java
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:**

```java
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.**

```java
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.

- **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
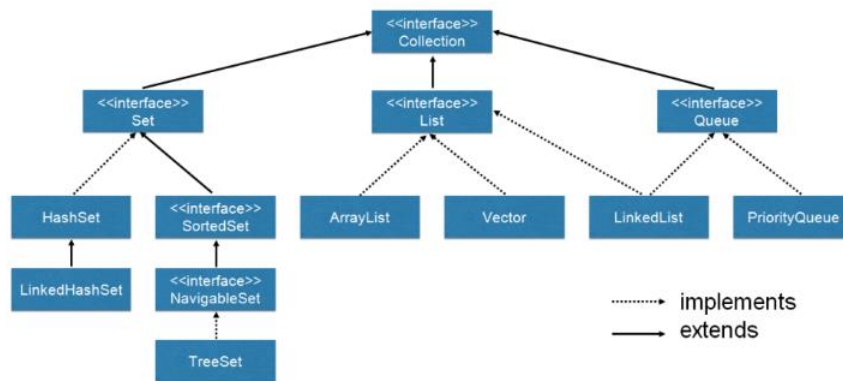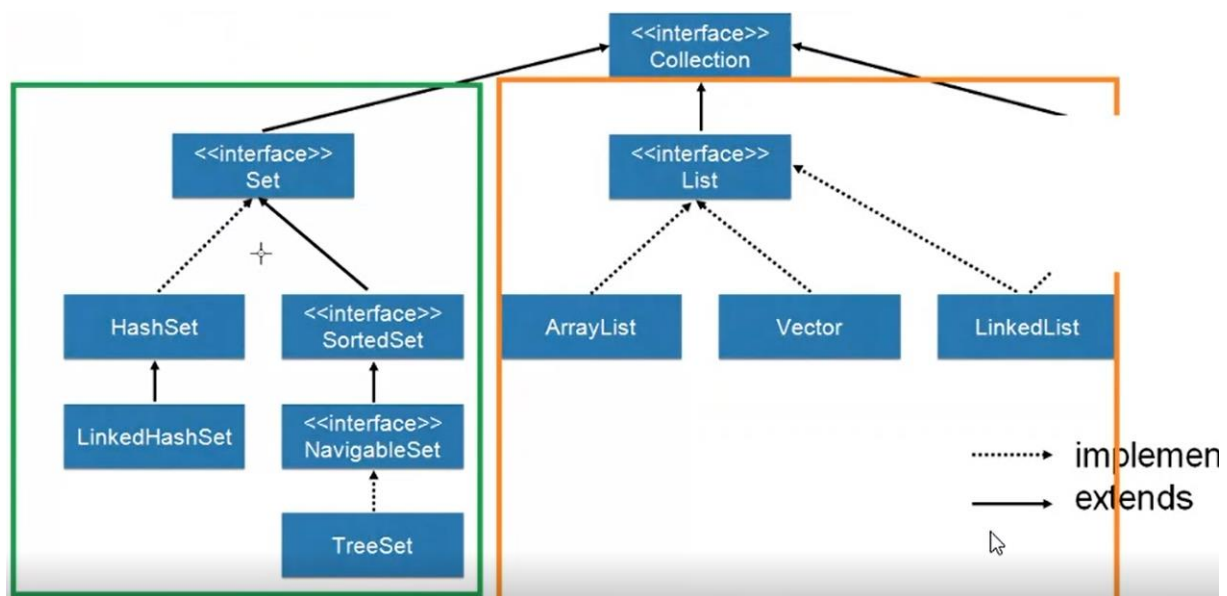   - Queue
    - Map
    - Sorted Map
    - Navigable Map

Figure 1: Class and Interface Heirarchy

❖ **List**
- It is child interface of collection.
- It is present in Java.util.Package.
- If we want to represent group of individual objects as single entity where duplicates are allowed and insertion order is preserved then we should go for list.
- We can differentiates by using index.
- We can preserve insertion order by using index. Hence index plays important role. It will get the same sequence of element while retrieving the elements.
- It has three subclasses in java such as- <mark>**Array List, Linked List, Vector, etc.**</mark>
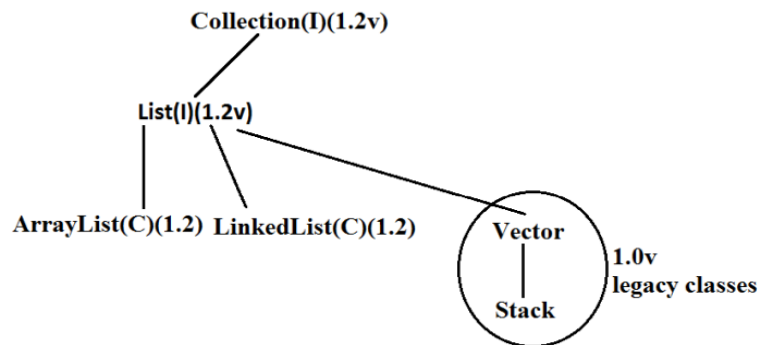
- Vector and Stack classes are re-engineered in 1.2 versions to implement List interface.

- **Array List**
  - The Array List extends implements the List interface.
  - 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-**

  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

  ArrayList a=new ArrayList(int initialcapacity);

  - Creates an empty ArrayList object with the specified initial capacity.

  ArrayList a=new ArrayList(collection c);

  - Here we are passing **the objects.**

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

```java
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
```

```java
        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
[]
```

<mark>**This is allowed but not recommended in Java. We are using Arraylist with Generics:**</mark>

```java
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:

```java
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.**

```java
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
```

• LinkedList

- It is child of collection.

- It is present in Java.util.Package.

- Insertion order is preserved.

- Underlying data structure is double linked list.

- 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 best choice if our frequent operation is insertion and deletion in middle.

- LinkedList is worst choice if our frequent operation is retrieval.
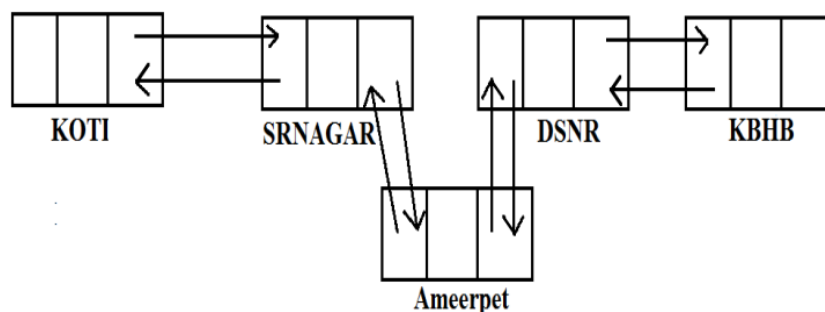
- Usually we can use LinkedList to implement Stacks and Queues.

- 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:

- **LinkedList l= new LinkedList();**
  Create the empty linked list object.
- **LinkedList ll= new LinkedList(Collection c);**
  Create the object for given collection.

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

```java
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:

- 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".

- Vector v=new Vector(int initialcapacity);

- Vector v=new Vector(int initialcapacity, int incrementalcapacity);

- 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++) {
```

```java
            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:

```java
import java.util.Stack;

public class StackList {

    public static void main(String[] args) {

        Stack stack = new Stack();
```

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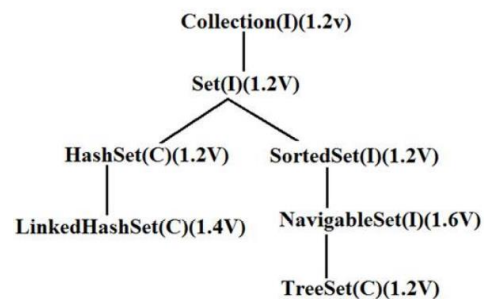
```
[J, M, K]
-1
[J, M]
[J, M]
```

- ## 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 RandomAccess.
- HashSet is best suitable, if our frequent operation is "Search".

Constructors:

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).

HashSet h=new HashSet (int initialcapacity);

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

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

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 Hash_Set:

```java
public class HashDemo {

public static void main(String[] args) {

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

        jk.add("Tanvir");

        jk.add("Shinde");
```

```
        jk.add("Kusumba");

        //Duplicate all are not allowded.

    jk.add("Tanvir");

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

Output on Console:

```
[Kusumba, Shinde, Tanvir]
```

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 LinkedHashSetDemo {

    public static void main(String[] args) {

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

            jk.add("Rupesh");

            jk.add("Shinde");

            jk.add("Tanvir");
```

```
            jk.add("Shinde");

            //Duplicate are not allowded...Inertion order preserved.

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

Output on Console:

```
[Rupesh, Shinde, Tanvir]
```

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.
- Heterogenous objects are not allowed.
- If we are trying to insert the heterogenous objects then will get run time exception saying classcastexception.
- Null insertion is not allowed, if we are trying to insert it then will get run time error as NullPointerException.

Constructor-

TreeSet ts= new TreeSet();

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

TreeSet ts= new TreeSet(Comparator c);

Create empty treeset object where elements will be inserted according to customized sorting order.

TreeSet t= new TreeSet(SortedSet s);

TreeSet t= new TreeSet(Collection c);

Java Program on Tree_Set:

```java
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.

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

> public boolean hasMoreElements();
>
> public Object nextElement()

Java program on enumeration cursor:

```java
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]
```

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.

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.

    public boolean hasNext();

    public object next();

    public void remove();

Java program on Iterator cursor:

```java
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]
```

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.

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=l.listIterator();

    (l is any List object)

ListIterator interface defines the following 9 methods.

    public boolean hasNext();

    public Object next(); forward

    public int nextIndex();

    public boolean hasPrevious();

    public Object previous(); backward

    public int previousIndex();

    public void remove();

    public void set(Object new);

    public void add(Object new)

Java program on List_Iterator cursor:

```java
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();
                    }

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

    }
```

Output on Console:

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

Comparison of Enumeration, Iterator and ListIterator?

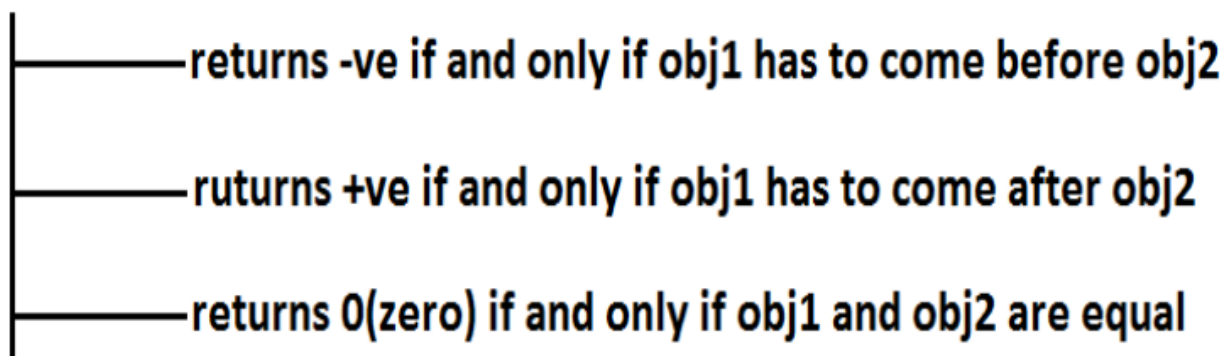| Property	Enumeration	Iterator	ListIterator
Is it legacy?	Yes	No	No
It is applicable for?	Only legacy classes.	Applicable for any collection object.	Applicable for only list objects.
Movement?	Single direction cursor(forward)	Single direction cursor(forward)	Bi-directional
How to get it?	By using elements() 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.

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:

```java
package collectionInterface;

public class Test {

        public static void main(String[] args) {

                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:

```
-25
0
-1
```

Comparator interface:

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

  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

  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.

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

```java
package collectionInterface;

import java.util.TreeSet;

public class TreeTestDemo {

    public static void main(String[] args) {

        TreeSet t = new TreeSet(new MyComparator());

        t.add(10);

        t.add(5);

        t.add(20);

        t.add(8);

        System.out.println(t);

    }
}
```

Output on Console:

[20, 10, 8, 5]


```java
package collectionInterface;

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)

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.
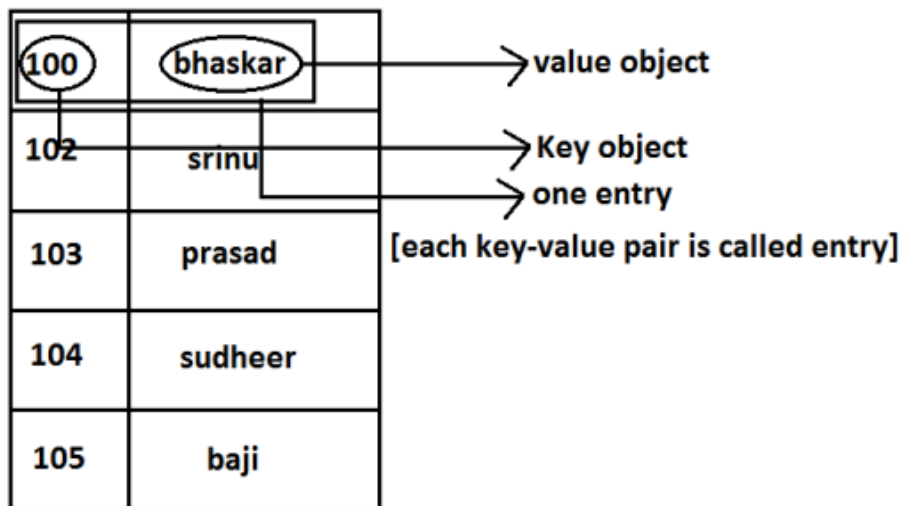
Disadvantage of synchronization?

- 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 4. Each key-value pair is called "one entry".

Map(I)(1.2V)

HashMap(C)(1.2V)   IdentityHashMap(C)(1.4V)   SortedMap(I)(1.2V)   Dictionary(AC)

LinkedHashMap(C)(1.4V)   WeakHashMap(C)(1.2V) NavigableMap(I)(1.6V)   Hashtable(C)

TreeMap(C)(1.2V)   Properties(C)
1.0V

Diagram:

100	bhaskar	→ value object
102	srinu	→ Key object → one entry
103	prasad	[each key-value pair is called entry]
104	sudheer	
105	baji	
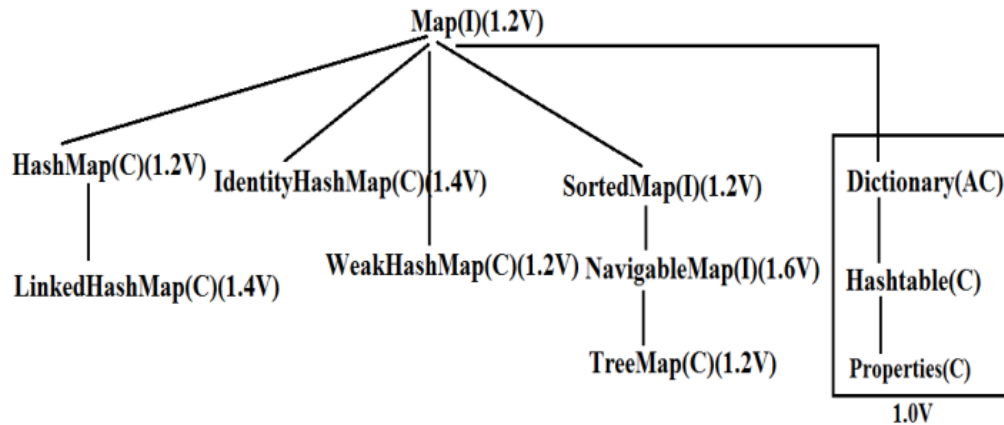
- Map interface is not child interface of Collection and hence we can't apply Collection interface methods here.

- **HashMap:**
    - A HashMap is class which implements the Map interface
    - 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.
    - Return type of put method is Object.

- **Java Program on HashMap:**

```java
import java.util.HashMap;

public class HashMapDemo {

        public static void main(String[] args) {

                HashMap hashMap = new HashMap();
                hashMap.put(10, "ashok");
                hashMap.put(11, "ram");

                System.out.println(hashMap);

        }

}
```

Output on Console:

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

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

```java
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}
```

TreeMap-

- The TreeMap is a class which implements NavigableMap interface which is the sub-interface of SortedMap.
- It stores values based on key
- It is ordered but in an Ascending manner
- Keys should be unique
- It cannot have null key at run time but can have null values because the interpreter will not understand how to sort null with other values.

```java
package collectionInterface;

import java.util.TreeMap;

public class HashMapDemo3 {

    public static void main(String[] args) {

        TreeMap treeMap = new TreeMap();
        treeMap.put(10, "Ajay");
        treeMap.put(11, "ram");
        treeMap.put(12, "shyam");

        System.out.println(treeMap);

    }

}
```

Output on Console:

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

Hashtable-

- Hashtable is a class which implements Map interface and extends Dictionary class.

- It stores values based on key

- It is unordered and the key should be unique

- It cannot have null keys or null values. It gives runtime error if we try to add any null keys or values but will not show an error at compile time.

- It has synchronised methods and slower than hashmap.

```java
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-

{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

```java
package collectionInterface;
```

```java
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));

            /*
             * get method used to get the respective value of key.
             */
        }

    }


}
```
Output on Console:

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


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

public class HashMapDemo5 {

    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();
        Iterator<Integer> itr = s.iterator();
        while (itr.hasNext()) {
```

```
            int i = itr.next();
            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
```

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

Exception Handling :

Exception:

- An Exception is an unwanted or unexpected condition which disturbs our normal flow of execution.
- Once Exception occured 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 the 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.

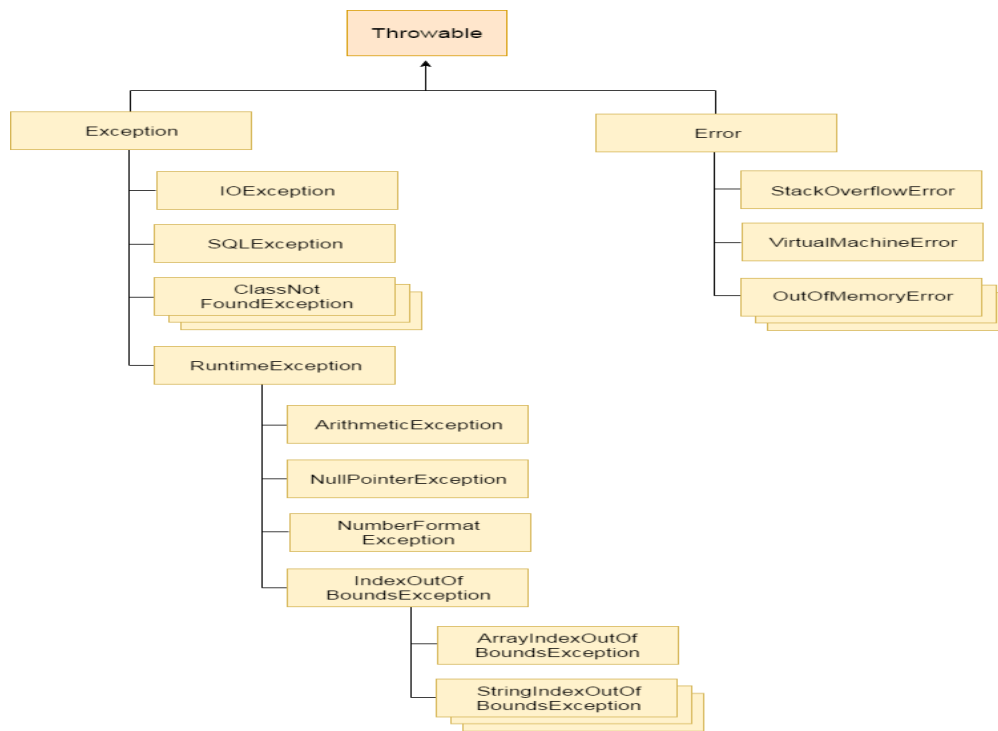**Depending on Hierarchy, Exceptions are divided into 2 types**

- 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.

(Or)

- Exception classes which are directly inheriting Exceptionclass 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 -FileNotFoundException etc.**

```
                        Throwable
                            ↑
        ┌───────────────────┴───────────────────┐
    Exception                                  Error
        │  IOException                            │  StackOverflowError
        │  SQLException                           │  VirtualMachineError
        │  ClassNot                               │  OutOfMemoryError
        │  FoundException
        │  RuntimeException
                │  ArithmeticException
                │  NullPointerException
                │  NumberFormat
                │  Exception
                │  IndexOutOf
                │  BoundsException
                        │  ArrayIndexOutOf
                        │  BoundsException
                        │  StringIndexOutOf
                        │  BoundsException
```

Unchecked Exception

- Exception which are checked(identified or found out) during Runtime or execution time, such type of exception are called as Unchecked Exceptions.
- Incase of Unchecked Exception our program will atleast compiles successfully.
- Unchecked Exceptions are also called as Runtime Exceptions.
- RuntimeExceptionclass is a super class to all UncheckedException classes.
- **Examples(Classes) of Unchecked Exceptions are :-**
  - ArithmeticException
  - ArrayIndexOutOfBoundsException
  - NullPointerException
  - StringIndexOutofBoundsException

- ClassCastException
- 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 :**

- StackoverFlowError
- VirtualMemoryError
- 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:

```java
public class Test3 {

    public static void main(String[] args) {

        int i =10;
        int j=0;
```

```java
        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:

```java
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:

```java
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 exception.ThrowKeyword.main(ThrowKeyword.java:12)

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 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.

Can we write only try block without catch block ?

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

  Ex: Exception e=new ArithmeticException()

  Ex: Throwable e1=new ArithmeticException ()