```
import java.util.*;

Reversing the Elements of List:
      public static void reverse(List l);


reverse() VsreverseOrder():
=> We can Use reverse() to Reverse Order of Elements of List.
=> We can Use reverseOrder() to get Reversed Comparator.

Comparator c1 = Collecctions.reverseOrder(Comparator c);
           |                                        |
   Descending Order                          Ascending Order


eg#1.

public class Test
{
      public static void main(String[] args)
      {
            ArrayList al = new ArrayList();
            al.add(15);
            al.add(0);
            al.add(20);
            al.add(10);
            al.add(5);

            Comparator c1 = new MyComparator();

            //Collections
            Collections.sort(al,c1);
            System.out.println("After Sorting   :: "+al);//[20, 15, 10, 5, 0]

            Collections.reverse(al);
            System.out.println("After reversing :: "+al);//[0,5,10,15,20]

            Comparator c2 =Collections.reverseOrder(c1);
            Collections.sort(al,c2);
            System.out.println("ReverseOrder Sorting  :: "+al);//[0,5,10,15,20]
      }
}
class MyComparator implements Comparator
{
      @Override
      public int compare(Object obj1,Object obj2)
      {
            //Sort:: Descending order
            Integer i1 = (Integer) obj1;
            Integer i2= (Integer) obj2;
            return -i1.compareTo(i2);
      }
}


Arrays
======
=> Arrays Class is an Utility Class to Define Several Utility Methods for Array
Objects.
```

Sorting Elements of Array:
1) public static void sort(primitive[] p); To Sort According to Natural Sorting Order.
2) public static void sort(Object[] o); To Sort According to Natural Sorting Order.
3) public static void sort(Object[] o, Comparator c);To Sort According to Customized Sorting Order.

Note:
=> For Object Type Arrays we can Sort According to Natural Sorting Order OR Customized Sorting Order.
=> But we can Sort primitive[] Only Based on Natural Sorting.

eg#1.

```
import java.util.*;
public class Test
{
	public static void main(String[] args)
	{
		int[] a= {10,5,20,11,6};
		System.out.println("Primitive Array before Sorting...");
		for (int data: a )
		{
			System.out.print(data+"\t");//10	5	20	11	6
		}
		System.out.println();

		//public static void sort(int[]);
		Arrays.sort(a);
		System.out.println("Primitive Array after Sorting...");
		for (int data: a )
		{
			System.out.print(data+"\t");//5	6	10	11	20
		}

		System.out.println("\n");


		String[] names = {"sachin","saurav","dhoni","kohli","azarudin"};
		System.out.println("Object Array before Sorting...");
		for (String data: names )
		{
			System.out.print(data+"\t");//sachin saurav dhoni kohli azarudin
		}
		System.out.println();

		//public static void sort(java.lang.Object[]);
		Arrays.sort(names);
		System.out.println("Object Array after Sorting...");
		for (String data: names )
		{
			System.out.print(data+"\t");//azarudin dhoni kohli sachin saurav
		}

		System.out.println();

		//public static <T> void sort(T[], java.util.Comparator<? super T>)
```

```
            Arrays.sort(names,new MyComparator());
            System.out.println("Object Array after Sorting using Comparator...");
            for (String data: names )
            {
                  System.out.print(data+"\t");//saurav  sachin  kohli   dhoni
azarudin
            }
            System.out.println();
      }
}
class MyComparator implements Comparator
{
      @Override
      public int compare(Object obj1,Object obj2)
      {
            //Sort:: Descending order
            String s1 = obj1.toString();
            String s2 = obj2.toString();
            return -s1.compareTo(s2);
      }
}

Output
Primitive Array before Sorting...
10      5       20      11      6
Primitive Array after Sorting...
5       6       10      11      20

Object Array before Sorting...
sachin  saurav  dhoni   kohli   azarudin
Object Array after Sorting...
azarudin        dhoni   kohli   sachin  saurav
Object Array after Sorting using Comparator...
saurav  sachin  kohli   dhoni   azarudin



Searching the Elements of Array
================================
1) public static int binarySearch(primitive[] p, primitive target);
   If the Primitive Array Sorted According to Natural Sorting Order then we have to
Use this
   Method.
2) public static int binarySearch(Object[] a, Object target);
   If the Object Array Sorted According to Natural Sorting Order then we have to
Use this
   Method.
3) public static intbinarySearch(Object[] a, Object target, Comparator c);
   If the Object Array Sorted According to Comparator then we have to Use this
Method.

Note: All Rules of Array Class binarySearch() are Exactly Same as Collections Class
binarySearch().


eg#2.
import java.util.*;
```

```
//Client Code
public class Test
{
	public static void main(String[] args)
	{
		int[] a= {10,5,20,11,6};

		//public static void sort(int[]);
		//5,6,10,11,20
		Arrays.sort(a);//DNS

		//public static int binarySearch(int[], int);
		//success(key found)     : return index
		//failure(key not found) : return insertion point
		System.out.println(Arrays.binarySearch(a,20));//4
		System.out.println(Arrays.binarySearch(a,14));//-5

		System.out.println();
		String names[]={"sachin","saurav","kohli","dhoni","azarudin"};

		//public static void sort(object[]);
		Arrays.sort(names);//azarudin,dhoni,kohli,sachin,saurav

		//public static int binarySearch(Object[],Object);
		//success(key found)     : return index
		//failure(key not found) : return insertion point
		System.out.println(Arrays.binarySearch(names,"kohli"));//2
		System.out.println(Arrays.binarySearch(names,"raina"));//-4

		System.out.println();

		//saurav sachin kohli dhoni azarudin
		Arrays.sort(names,new MyComparator());

		//public static int binarySearch(Object[],Object,Comparator c);
		//success(key found)     : return index
		//failure(key not found) : return insertion point
		System.out.println(Arrays.binarySearch
				(names,"kohli",new MyComparator()));//2
		System.out.println(Arrays.binarySearch
				(names,"raina",new MyComparator()));//-3
	}
}
class MyComparator implements Comparator
{
	@Override
	public int compare(Object obj1,Object obj2)
	{
		//Sort:: Descending order
		String s1 = obj1.toString();
		String s2 = obj2.toString();
		return -s1.compareTo(s2);
	}
}

Conversion of Array to List
===========================
Note: To convert the Collection to Array we have a method called
		Object[] toArray()
```

=> Arrays Class contains asList() for this
   public static List asList(Object[] a);
=> Strictly Speaking this Method won't Create an Independent List Object, Just we are
   Viewing existing Array in List Form.

=> By using Array Reference if we Perform any Change Automatically that Change will be
   reflected to List Reference.
=> Similarly by using List Reference if we Perform any Change Automatically that Change will
   be reflected to Array.
=> By using List Reference if we are trying to Perform any Operation which Varies the Size
   then we will get Runtime Exception Saying UnsuportedOperationException.

eg#1.

```
import java.util.*;

//Client Code
public class Test
{
      public static void main(String[] args)
      {
            //Array : size is fixed and it holds only homogenous type elements
            String arr[] = {"A","Z","B"};


            //Converting Array to List[Read only Array]
            List list = Arrays.asList(arr);
            System.out.println(list);//[A,Z,B]

            System.out.println();

            System.out.println("Performing operation through Array");
            arr[0] = "K";
            System.out.println(list);//[K,Z,B]

            System.out.println();

            System.out.println("Performing operation through List");
            list.set(1,"L");
            System.out.println(list);

            //Updating the element to List(indirectly to an Array)
            list.set(1,new Integer(10));
            System.out.println(list);//ArrayStoreException

            //Adding the element to List(indirectly to an Array)
            list.add("sachin");
            System.out.println(list);//UnSupportedOperationException

            //Remove the element from List(indirectly to an Array)
            list.remove(2);
            System.out.println(list);//UnSupportedOperationException

      }
}
```

ConcurrentCollections
=====================
Objective
1. Majority of the Collection classes are not Threadsafety due to which there is a possiblity
   of "DataIncosistency" problem, so these collection classes are not ThreadSafe.
2. Already existing ThreadSafe Collection classes are
      a. Vector
      b. Hashtable
        c. synchronizedList(),synchronizedSet(),synchronizedMap() of Collections class performance
         not upto the mark because it increases the waiting time for other threads even if
         they want to just do read operation.

eg#1.
```
import java.util.ArrayList;
import java.util.Iterator;
public class Test {
      public static void main(String[] args) {
            ArrayList al = new ArrayList();
                  al.add("A");
                  al.add("B");
                  al.add("C");
            Iterator itr = al.iterator();
            while (itr.hasNext()){
                  String s = (String)itr.next();
                  System.out.println(s);
                  al.add("D");//java.util.ConcurrentModificationException
            }
      }
}
```

3. In Normal Collections,Simultaneoulsy if one Thread is performing read operation ,other thread can't perform write operation if we try
   to do it would result in "ConccurentModificatinException"
    So normal Collections are not suitable for "MultiThreading Applications".

eg#2.
```
import java.util.*;

//Client Code
public class Test extends Thread
{
      static ArrayList l=new ArrayList();

      @Override
      public void run()
      {
            //logic of a thread
            try
            {
                  Thread.sleep(2000);
            }
            catch (InterruptedException ie)
            {
                  ie.printStackTrace();
```

```
            }

            System.out.println("Child Thread updating list");
            l.add("B");

      }


      public static void main(String[] args) throws InterruptedException{

            l.add("A");
            l.add("B");
            l.add("C");

            Test t=new Test();
            t.start();

            //logic of main thread
            Iterator itr=l.iterator();
            while(itr.hasNext()){
                 String data=(String)itr.next();
                 System.out.println(
                             "Main Thread iterating list and the object is :
"+data);
                   Thread.sleep(3000);
            }
            System.out.println(l);

    }
}
```

```
Output
Main Thread iterating list and the object is : A
Child Thread updating list
Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:909)
        at java.util.ArrayList$Itr.next(ArrayList.java:859)
        at Test.main(Test.java:39)
Press any key to continue . . .
```

To resolve the above mentioned problems SUNMS introduced a new concpet in 1.5v called "ConcurrentCollections".

KeyPoints of ConcurrentCollections
==========

1) Concurrent Collections are Always Thread Safe.
2) When compared with Traditional Thread Safe Collections Performance is More because of different Locking Mechanism
   (segment locking/bucket locking mechanism).
3) While One Thread interacting Collection the Other Threads are allowed to Modify Collection in Safe Manner
   (best suited on Scalable Threading Application).

Note: ConcurrentCollections -> Special package java.util.concurrent.*;

Most important ConcurrentCollections
==========

1.ConcurrentHashMap
2.CopyonWriteArrayList
3.CopyonWriteArraySet

ConcurrentHashMap
=================
    Map(I)
     |extends
 ConcurrentMap(I)
     |
     |implements
     |
 ConcurrentHashMap(C)

Normal map => put(Object obj)
                    |=> If the key already present, then it will update the value for the
                        corresponding key and old value will be returns.

ConcurrentMap => putIfAbsent(Object obj)
                                |=> If the key is already present,then it will not update
                                the key with new value.

eg#1.
```
import java.util.concurrent.*;
public class Test
{
      public static void main(String[] args){
            ConcurrentHashMap chm = new ConcurrentHashMap();
            chm.put(10,"sachin");
            chm.put(10,"afridi");
            System.out.println(chm);//{10=afridi}
            chm.putIfAbsent(10,"sachin");
            System.out.println(chm);//{10=afridi}

    }
}
```

2.
Normal map => remove(Object obj)
                    |=> If the key already present, then it will remove that particular
                        entry from the Map.

ConcurrentMap => remove(Object Key,Object value)
                                |=> If both key and value matches only then that particular
                                  entry will be removed from the Map.

eg#1.
```
import java.util.concurrent.*;
```

```
//Client Code
public class Test
{
      public static void main(String[] args){
            ConcurrentHashMap chm = new ConcurrentHashMap();
            chm.put(10,"sachin");
            chm.remove(10,"afridi");
            System.out.println(chm);//{10=sachin}

            chm.remove(10,"sachin");
            System.out.println(chm);//{}
    }
}
```

3.

```
Normal map   => replace(Object key,Object value)
                    |=> If the key already present, then update a new value.

ConcurrentMap => replace(Object Key,Object value,Object newValue)
                                |=> If both key and value matches only then update
the
                                new value.
```

eg#1.

```
import java.util.concurrent.*;
//Client Code
public class Test
{
      public static void main(String[] args){
            ConcurrentHashMap chm = new ConcurrentHashMap();
            chm.put(10,"sachin");
            chm.replace(10,"afridi","messi");
            System.out.println(chm);//{10=sachin}

            chm.replace(10,"sachin","messi");
            System.out.println(chm);//{10=messi}
    }
}
```

ConcurrentHashMap

==========

=> Underlying Data Structure is Hashtable.

=> ConcurrentHashMap allows Concurrent Read and Thread Safe Update Operations.

=> To Perform Read Operation Thread won't require any Lock. But to Perform Update Operation Thread requires Lock but it is the Lock of Only a Particular Part of Map (Bucket Level Lock).

=> Instead of Whole Map Concurrent Update achieved by Internally dividing Map into Smaller Portion which is defined by Concurrency Level.

=> The Default Concurrency Level is 16.

=> That is ConcurrentHashMap Allows simultaneous Read Operation and simultaneously 16 Write (Update) Operations.

=> null is Not Allowed for Both Keys and Values.

=> While One Thread iterating the Other Thread can Perform Update Operation and ConcurrentHashMap Never throw ConcurrentModificationException.

Constructors:

1) ConcurrentHashMap m = new ConcurrentHashMap();

```
      Creates an Empty ConcurrentHashMap with Default Initial Capacity 16 and
Default Fill
      Ratio 0.75 and Default Concurrency Level 16.
2) ConcurrentHashMap m = new ConcurrentHashMap(int initialCapacity);
3) ConcurrentHashMap m = new ConcurrentHashMap(int initialCapacity, float
fillRatio);
4) ConcurrentHashMap m = new ConcurrentHashMap(int initialCapacity, float
fillRatio,
                                                  int concurrencyLevel);
5) ConcurrentHashMap m = new ConcurrentHashMap(Map m);

eg#1.
import java.util.concurrent.ConcurrentHashMap;
class Test {
public static void main(String[] args) {
      ConcurrentHashMap m = new ConcurrentHashMap();
      m.put(101, "A");
      m.put(102, "B");

      m.putIfAbsent(103, "C");
      m.putIfAbsent(101, "D");

      m.remove(101, "D");
      m.replace(102, "B", "E");

      System.out.println(m); //{103=C, 102=E, 101=A}
  }
}

eg#2.
import java.util.concurrent.*;
import java.util.*;

//Client Code
public class Test extends Thread
{
      static ConcurrentHashMap m = new ConcurrentHashMap();

      @Override
      public void run()
      {
            try {
                  Thread.sleep(2000);
            }
            catch (InterruptedException e) {}
            System.out.println("Child Thread updating Map");
            m.put(103, "C");
      }

      public static void main(String[] args) throws Exception
      {
            m.put(101, "A");
            m.put(102, "B");

            Test t =new Test();
            t.start();


            Set s = m.keySet();
```

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

            while (itr.hasNext())
            {
                  Integer I1 = (Integer) itr.next();
                  System.out.println("Main Thread iterating and Current Entry
is:"+I1+"..............."+m.get(I1));

                  Thread.sleep(3000);
        }
            System.out.println(m);
    }
}
Output
Main Thread iterating and Current Entry is:101...............A
Child Thread updating Map
Main Thread iterating and Current Entry is:102...............B
Main Thread iterating and Current Entry is:103...............C
{101=A, 102=B, 103=C}

=> Update and we won't get any ConcurrentModificationException.
=> If we Replace ConcurrentHashMap with HashMap then we will get
ConcurrentModificationException.

Difference b/w HashMap and ConcurrentHashMap
============================================

HashMap           => Not Thread Safe
ConcurrentHashMap => Thread Safe


HashMap           => Performance is high as Threads are not required to wait to
operate.
ConcurrentHashMap => Performance is low as Threads are required to operate.

HashMap           => One Thread while Opeating on HashMap,other thread are not
allowed to modify
                 if it tries to do it would result in
"ConcurrentModificationException".

ConcurrentHashMap => One Thread while Opeating on HashMap,other thread are  allowed
to modify
                     in safe manner, it wont throw
"ConcurrentModificationException".

HashMap           => Iterator of HashMap if FailFast.
ConcurrentHashMap => Iterator of ConcurrentHashMap is FailSafe.

HashMap           => null is allowed for both keys and values.
ConcurrentHashMap => null is not allowed for both keys and values.

HashMap           => Introduced in 1.2v
ConcurrentHashMap => Introduced in 1.5v


Difference b/w ConcurrentHashMap,synchronizedMap() and Hashtable
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

ConcurrentHashMap =>Thread safety without putting lock on entire Object lock at
```

bucket level
synchronizedMap() =>Thread safety by putting lock on entire Object
Hashtable   =>Thread safety by putting lock on entire Object

ConcurrentHashMap =>Mulitple Threads are allowed to operate on Object in same manner
synchronizedMap() =>Only One Thread is allowed to operate on Object
Hashtable   =>Only One Thread is allowed to operate on Object

ConcurrentHashMap =>Read operation can be performed without any Lock,to perform update/write we need Object level lock.
synchronizedMap() =>To perform read and update we need Object level lock
Hashtable   =>To perform read and update we need Object level lock

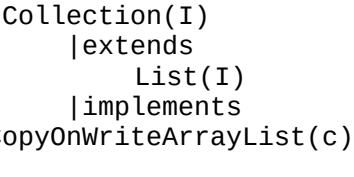
ConcurrentHashMap =>No ConcurrentModificationException
synchronizedMap() =>It would result in ConcurrentModificationException
Hashtable   =>It would result in ConcurrentModificationException

ConcurrentHashMap =>Iterator is FailSafe
synchronizedMap() =>Iterator is FailFast
Hashtable   =>Iterator is FailFast

ConcurrentHashMap =>null not allowed for both keys and values
synchronizedMap() =>null allowed for keys and values
Hashtable   =>null not allowed for both keys and values

ConcurrentHashMap =>JDK1.5V
synchronizedMap() =>JDK1.2V
Hashtable   =>JDK1.0V

CopyOnWriteArrayList
====================



package :: java.util.concurrent.CopyOnWriteArrayList(c)

=> It is a Thread Safe Version of ArrayList as the Name indicates CopyOnWriteArrayList
   Creates a Cloned Copy of Underlying ArrayList for Every Update Operation at Certain
   Point Both will Synchronized Automatically Which is taken Care by JVM Internally.

=> As Update Operation will be performed on cloned Copy there is No Effect for the Threads
   which performs Read Operation.

=> It is Costly to Use because for every Update Operation a cloned Copy will be Created.
   Hence CopyOnWriteArrayList is the Best Choice if Several Read Operations and Less
   Number of Write Operations are required to Perform.

=> Insertion Order is Preserved.
=> Duplicate Objects are allowed.

=> Heterogeneous Objects are allowed.
=> null Insertion is Possible.
=> It implements Serializable, Clonable and RandomAccess Interfaces.

=> While One Thread iterating CopyOnWriteArrayList, the Other Threads are allowed to
   Modify and we won't get ConcurrentModificationException. That is iterator is FailSafe.

=> Iterator of ArrayList can Perform Remove Operation but Iterator of CopyOnWriteArrayList can't
   Perform Remove Operation. Otherwise we will get RuntimeException Saying UnsupportedOperationException.

Constructors:
1) CopyOnWriteArrayList l = new CopyOnWriteArrayList();
2) CopyOnWriteArrayList l = new CopyOnWriteArrayList(Collection c);
3) CopyOnWriteArrayList l = new CopyOnWriteArrayList(Object[] a);

Methods
boolean addIfAbsent(Object o): The Element will be Added if and Only if List doesn't
                        contain this Element.

```
CopyOnWriteArrayList l = new CopyOnWriteArrayList();
l.add("A");
l.add("A");
l.addIfAbsent("B");
l.addIfAbsent("B");
System.out.println(l); //[A, A, B]
```

int addAllAbsent(Collection c): The Elements of Collection will be Added to the List if
                        Elements are Absent and Returns Number of Elements Added.

```
ArrayList l = new ArrayList();
  l.add("A");
  l.add("B");

CopyOnWriteArrayList l1 = new CopyOnWriteArrayList();
 l1.add("A");
 l1.add("C");
System.out.println(l1); //[A, C]

 l1.addAll(l);
System.out.println(l1); //[A, C, A, B]

ArrayList l2 = new ArrayList();
 l2.add("A");
 l2.add("D");
 l1.addAllAbsent(l2);
System.out.println(l1); //[A, C, A, B, D]
```

eg#1.

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

```
//Client Code
public class Test extends Thread
{
   static CopyOnWriteArrayList<String> cowal = new CopyOnWriteArrayList<String>();

   @Override
   public void run()
   {
            try {
                  Thread.sleep(2000);
            }
            catch (InterruptedException e) {}
            System.out.println("Child Thread updating List");
                  cowal.add("C");
    }
      public static void main(String[] args)throws Exception
      {
            cowal.add("A");
            cowal.add("B");

            Test t = new Test();
            t.start();

            Iterator itr = cowal.iterator();

            while (itr.hasNext()) {
                  String I1 = (String) itr.next();
                  System.out.println
                        ("Main Thread iterating and Current Object is :: "+I1);
                  Thread.sleep(3000);
       }
       System.out.println(cowal);
    }
}

Ouput
Main Thread iterating and Current Object is :: A
Child Thread updating List
Main Thread iterating and Current Object is :: B
[A, B, C]

=> In the Above Example while Main Thread iterating List Child Thread is allowed to
Modify and we won't get any
   ConcurrentModificationException.
=> If we Replace CopyOnWriteArrayList with ArrayList then we will get
ConcurrentModificationException.

eg#2.
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.Iterator;
class Test {
  public static void main(String[] args){
         CopyOnWriteArrayList l = new CopyOnWriteArrayList();
            l.add("A");
            l.add("B");
            l.add("C");
            l.add("D");
      System.out.println(l); //[A, B, C, D]
```

```
      Iterator itr = l.iterator();
      while (itr.hasNext()) {
            String s = (String)itr.next();
            if (s.equals("D"))
                  itr.remove();//RE: java.lang.UnsupportedOperationException
      }
      System.out.println(l);
    }
}
```

Iterator of CopyOnWriteArrayList can't Perform Remove Operation. Otherwise we will get RuntimeException: UnsupportedOperationException.

```
eg#3.
import  java.util.concurrent.CopyOnWriteArrayList;
import  java.util.Iterator;
class Test {
  public static void main(String[] args) {
      CopyOnWriteArrayList l = new CopyOnWriteArrayList();
            l.add("A");
            l.add("B");
            l.add("C");

      Iterator itr = l.iterator();
            l.add("D");
      while (itr.hasNext()) {
            String s = (String)itr.next();
            System.out.println(s); //[A,B,C]
      }
    }
}
```

Arraylist vs CopyOnWriteArrayList

++++++++++++++++++++++++++++++++

ArrayList => It is Not Thread Safe.

CopyOnWriteArrayList=> It is Thread Safe because Every Update Operation will be performed on Separate cloned Copy.

ArrayList => While One Thread iterating List Object, the Other Threads are Not allowed to Modify
List Otherwise we will get ConcurrentModificationException.

CopyOnWriteArrayList => While One Thread iterating List Object, the Other Threads are allowed to Modify List in Safe Manner and
we won't get ConcurrentModificationException.

ArrayList => Iterator is Fail-Fast.

CopyOnWriteArrayList => Iterator is Fail-Safe.

ArrayList => Iterator of ArrayList can Perform Remove Operation.

CopyOnWriteArrayList => Iterator of CopyOnWriteArrayList can't
Perform Remove Operation Otherwise we will
get RuntimeException: UnsupportedOperationException.

ArrayList => Introduced in 1.2 Version.

CopyOnWriteArrayList => Introduced in 1.5 Version.

Difference b/w CopyOnWriteArrayList,synchornizedList and Vector
++

CopyOnWriteArrayList
   We will get Thread Safety because Every Update Operation will be performed on Separate cloned Copy.
   At a Time Multiple Threads are allowed to Access/ Operate on CopyOnWriteArrayList.
   While One Thread iterating List Object, the Other Threads are allowed to Modify Map and we won't get ConcurrentModificationException.
   Iterator is Fail-Safe and won't raise ConcurrentModificationException.
   Iterator can't Perform Remove Operation Otherwise we will get UnsupportedOperationException.
   Introduced in 1.5 Version.

synchronizedList
  We will get Thread Safety because at a Time List can be accessed by Only One Thread at a Time.
  At a Time Only One Thread is allowed to Perform any Operation on List Object.
  While One Thread iterating , the Other Threads are Not allowed to Modify List. Otherwise we will get ConcurrentModificationException.
  Iterator is Fail-Fast and it will raise ConcurrentModificationException.
  Iterator canPerform Remove Operation.
  Introduced in 1.2 Version.

Vector
  We will get Thread Safety because at a Time Only One Thread is allowed to Access Vector Object.
  At a Time Only One Thread is allowed to Operate on Vector Object.
  While One Thread iterating, the Other Threads are Not allowed to Modify Vector. Otherwise we will get ConcurrentModificationException.
  Iterator is Fail-Fast and it will raise ConcurrentModificationException.
  Iterator can Perform Remove Operation.
  Introduced in 1.0 Version

CopyOnWriteArraySet
==================
Collection (I)
   |
 Set (I)
   |
CopyOnWriteArraySet (C)

=> It is a Thread Safe Version of Set.
=> Internally Implement by CopyOnWriteArrayList.
=> Insertion Order is Preserved.
=> Duplicate Objects are Not allowed.
=> Multiple Threads can Able to Perform Read Operation simultaneously but for Every

   Update Operation a Separate cloned Copy will be Created.
=> As for Every Update Operation a Separate cloned Copy will be Created which is Costly
   Hence if Multiple Update Operation are required then it is Not recommended to Use
   CopyOnWriteArraySet.
=> While One Thread iterating Set the Other Threads are allowed to Modify Set and we won't
   get ConcurrentModificationException.

=> Iterator of CopyOnWriteArraySet can Perform Only Read Operation and won't Perform
   Remove Operation. Otherwise we will get RuntimeException: UnsupportedOperatonException.

Constructors:
1) CopyOnWriteArraySets = new CopyOnWriteArraySet();
      Creates an Empty CopyOnWriteArraySet Object.
2) CopyOnWriteArraySet s = new CopyOnWriteArraySet(Collection c);
      Creates CopyOnWriteArraySet Object which is Equivalent to given Collection Object.

Methods:Whatever Methods Present in Collection and Set Interfaces are the Only Methods
      Applicable for CopyOnWriteArraySet and there are No Special Methods.

```
import java.util.concurrent.CopyOnWriteArraySet;
class Test {
      public static void main(String[] args) {
            CopyOnWriteArraySet s = new CopyOnWriteArraySet();
            s.add("A");
            s.add("B");
            s.add("C");
            s.add("A");
            s.add(null);
            s.add(10);
            s.add("D");
            System.out.println(s);//[A, B, C, null, 10, D]
       }
}
```

Differences between CopyOnWriteArraySet() and synchronizedSet()

CopyOnWriteArraySet()
==========
It is Thread Safe because Every Update Operation will be performed on Separate Cloned Copy.
While One Thread iterating Set, the Other Threads are allowed to Modify and we won't get ConcurrentModificationException.
Iterator is Fail Safe.
Iterator can Perform Only Read Operation and can't Perform Remove Operation
Otherwise we will get RuntimeException Saying UnsupportedOperationException.
Introduced in 1.5 Version.

synchronizedSet()
==========
It is Thread Safe because at a Time Only One Thread can Perform Operation.
While One Thread iterating, the Other Threads are Not allowed to Modify Set
Otherwise we will get ConcurrentModificationException.
Iterator is Fail Fast.
Iterator can Perform Both Read and Remove Operations.
Introduced in 1.7 Version.

Difference FailFast vs FailSafe Iterators
++
Fail Fast Iterator:While One Thread iterating Collection if Other Thread trying to

Perform
any Structural Modification to the underlying Collection then immediately Iterator Fails by
raising ConcurrentModificationExcepion. Such Type of Iterators are Called Fail Fast Iterators.

```
import java.util.ArrayList;
import java.util.Iterator;
class Test {
      public static void main(String[] args) {
            ArrayList l = new ArrayList();
            l.add("A");
            l.add("B");
            Iterator itr = l.iterator();//FailFast Iterator
            while(itr.hasNext()) {
                  String s = (String)itr.next();
                  System.out.println(s); //A
                  l.add("C"); // java.util.ConcurrentModificationException
            }
      }
}
```

Note: Internally Fail Fast Iterator will Use Some Flag named with MOD to Check underlying
Collection is Modified OR Not while iterating.

Fail Safe Iterator:
=> While One Thread iterating if the Other Threads are allowed to Perform any Structural
   Changes to the underlying Collection, Such Type of Iterators are Called Fail Safe Iterators.

=> Fail Safe Iterators won’t raise ConcurrentModificationException because Every Update
   Operation will be performed on Separate cloned Copy.

```
import java.util.concurrent.CopyOnWriteArraySet;
import java.util.Iterator;
class Test {
      public static void main(String[] args) {
            CopyOnWriteArraySet l = new CopyOnWriteArraySet();
            l.add("A");
            l.add("B");
            Iterator itr = l.iterator();//FailSafe Iterator
            while(itr.hasNext()) {
                  String s = (String)itr.next();
                  System.out.println(s); //A B
                  l.add("C");
            }
            System.out.println(l);//[A,B,C]
      }
}
```