Concurrent Collections

Why we go for Concurrent Collections?

Let's have a look on traditional Collections

- Most of the collections are not thread safe. So if multiple threads are allowed on the same collection there may be a chance of data inconsistency.
- Even in traditional collections there are some thread safe collections like vector, Hashtable, SynchronizedList(), SynchronizedSet(), SynchronizedMap(). So using these concepts we can achieve thread safety but how we are getting it is a problem in itself. In all the above thread safe collections only one thread is allowed at a time. Even for read operation threads has to wait. For any operation total collection object will be locked. It increases waiting time and hence lowers application performance.
- While one thread iterating a collection other thread trying to modify the same collection immediately iteration fails stating ConcurrentNodificationException.

  Due to all these issue traditional collections are not suitable for multithreaded scalable applications. Suppose in our application lakhs of thread working at a time there may be a chance for data inconsistency (point-1), Performance issue (point-2) and structural modification exception(point-3).
  To overcome these problem we should go for Concurrent Collections.

Let's understand concurrent modification by an example

```
import java.util.ArrayList;
import java.util.Iterator;
/*
 * In this example main thread is iterating on the same
 moment child thread is trying to modify so
 ConcurrentModificationException occurs.
 */

public class MyThread extends Thread {
	static ArrayList<String> alphabetList=new ArrayList();
	@Override
	public void run()
	{
		try {
			Thread.sleep(5000);
			alphabetList.add("E");
			System.out.println("Child Thread is trying to update
alphabetList");
		} catch (InterruptedException e) {

		}
	}

	public static void main(String[] args) throws InterruptedException {
		/*
		 main thread is adding elements in alphabetList
		 */
		alphabetList.add("A");
		alphabetList.add("B");
		alphabetList.add("C");
		alphabetList.add("D");
		MyThread t=new MyThread();
		t.start();
		Iterator<String> itr=alphabetList.iterator();
		while(itr.hasNext())
		{
			String alphabet=(String)itr.next();
			System.out.println("main thread is iterating the alphabetList 
and current object is : "+alphabet);
			Thread.sleep(2000);
		}
		
		System.out.println(alphabetList);
		
	}

}
```

Output-

```
main thread is iterating the alphabetList and current object is : A
main thread is iterating the alphabetList and current object is : B
main thread is iterating the alphabetList and current object is : C
Child Thread is trying to update alphabetList
Exception in thread "main" java.util.ConcurrentModificationException
	at java.base/java.util.ArrayList$Itr.checkForComodification(Unknown Source)
	at java.base/java.util.ArrayList$Itr.next(Unknown Source)
	at MyThread.main(MyThread.java:36)
```

Traditional Collection vs Concurrent Collection

Traditional Collection	**Concurrent Collection**
Not thread safe	Thread safe
Performance low	Performance relatively high (Now question is how—if it is also thread safe like vectors the performance is high how **Answer is different locking mechanism**)
Concurrent modification not allowed. Concurrent modification exception is thrown.	Concurrent modification allowed. It happens in safer way.

All concurrent Collections are placed **in java.util.concurrent** package.

Examples for Concurrent Collections-

- Concurrent HashMap
- CopyOnWriteArrayList
- CopyOnWriteArraySet

Concurrent HashMap

It implements ConcurrentMap Interface which extends Map Interface. All methods present in Map are available in Concurrent HashMap. In Concurrent HashMap three extra methods have been introduced.

i. Object putIfAbsent(Object key, Object value)

Demonstration:

```
ConcurrentHashMap<Integer, String> conHashMap=new ConcurrentHashMap<>();

		/*Normal put method of Map while duplicate key entry*/
		conHashMap.put(101, "Sanjay");
		conHashMap.put(101, "Ranjeev");
		System.out.println(conHashMap);//Output is {101=Ranjeev} old value 
is overided

		/*putIfAbsent method checks if key not exists then only put
		 * if exits then don't override
		 * */
		conHashMap.putIfAbsent(101, "Rama");//will retain old value
		conHashMap.putIfAbsent(102, "Changed");// key 102 does not exits so 
new entry will be added.
			System.out.println(conHashMap);
```

ii. Boolean remove(Object key, Object value)

```
conHashMap.put(101, "Sanjay");
		conHashMap.put(101, "Ranjeev");
		//conHashMap.remove(101);// it will remove the entry with 101 and 
map is now empty
		conHashMap.remove(101, "Ranjeev");// Map has only one entry with 
Ranjeev which will be removed.
			System.out.println(conHashMap);
```

iii. Boolean replace (Object key, Object oldValue, Object newValue)

Will be replaced only if both key and value matched. In traditional map key is compared only.

```
conHashMap.put(103, "JP");
		conHashMap.replace(103, "PJ", "New JP");//no replacement
		conHashMap.replace(103, "JP", "JP New");//replacement happens
```

How Concurrent HashMap works internally and how it differs from Hashtable?

For reading operation in HashMap or Hashtable it requires map/table level lock so only one thread can read at a time but **in Concurrent HashMap reading operation does not requires any lock so any number of threads can operate at a time**. To read/update it requires segment/Bucket level locking.

Default initial capacity of Concurrent HashMap-16

So total 16 buckets will be there. While reading/updating in concurrent HashMap map level lock is not required. It requires bucket level locking so at a time 16 threads can do update operation.

Concurrent HashMap is internally divided into 16 parts (Buckets).

Segment level locking- Total Concurrent HashMap will be divided into a number of segments default is 16. And this number (16) is called concurrency level and for each bucket a separate lock will be maintained. Most of the time number of bucket and concurrency level is same that's why one bucket one lock we can say.

Consider three scenarios

- Initial capacity is 16 and concurrency level is also 16
  Here initial capacity is 16. Concurrency level is 16 means 16 locks we have. So every bucket requires one lock and this is called bucket level locking.
- Initial capacity is 16 and concurrency level is 8
  Here concurrency level is 8 so for every two buckets (Two buckets is one Segment here) one lock will be maintained. This is called segment level locking. In two buckets (one segment) only one thread is allowed to update operation.
- Initial capacity is 16 and concurrency level is 32
  Here every bucket will have 2 locks

Summary : In Concurrent HashMap any number of threads allowed at a time to read. To write/update it is equal is number of concurrency level. Default is 16.

Program-

```
package concurrentCollections;

import java.util.Iterator;
import java.util.Map.Entry;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;

public class CHMUpdate extends Thread{
	static ConcurrentHashMap<Integer, String> chm=new ConcurrentHashMap();
	public void run()
	{
		System.out.println("Child thread is updating concurrent HashMap");
		chm.put(200, "Black");
		try {
			Thread.sleep(2000);
		} catch (InterruptedException e) {

			e.printStackTrace();
		}
	}


	public static void main(String[] args) {
		chm.put(100, "Red");
		chm.put(300, "Yellow");
		chm.put(400, "Green");
		chm.put(900, "Grey");
		chm.put(230, "cyan");
		Set chmKeySet=chm.keySet();
		CHMUpdate t=new CHMUpdate();
		t.start();
		Iterator<Entry<Integer, String>> itr=chm.entrySet().iterator();
		while(itr.hasNext())
		{
			Entry entry=(Entry)itr.next();
			System.out.println(entry.getKey());
		}

		System.out.println("-----------------------------------");
		System.out.println(chm);
```

```
            }
}
```

```
Output First Time-
400
Child thread is updating concurrent HashMap
100
900
230
200
300
------------------------------------
{400=Green, 100=Red, 900=Grey, 230=cyan, 200=Black, 300=Yellow}
```

```
Output Second Time-
Child thread is updating concurrent HashMap
400
100
900
230
200
300
------------------------------------
{400=Green, 100=Red, 900=Grey, 230=cyan, 200=Black, 300=Yellow}
```

Though Program is same output may be different some of the times-

HashMap updates the cells (Buckets). There may be two scenarios

In Output 1 {200=Black} is not available in Iterator. Reason is after crossing the cell {200=Black } entry is added into the bucket. And Iterator always iterates in forward direction only.

In output 2 {Black=200} is available in the iterator reason is before crossing the bucket itself entry has been updated.

So conclusion is while doing parallel updating there is no guarantee that entry will be available in the iterator but when we will print the Concurrent HashMap we will get updated entry.

Difference between HashMap and Concurrent HashMap

HashMap	**Concurrent HashMap**
Not Thread Safe	Thread Safe
Performance is high as any number of threads Allowed to read/write operation at a time.	Relatively performance is low. Any number of threads are allowed to read but for write only 16 threads are allowed at a time. When 17th thread comes it has to wait so performance decreased.
Concurrent modification not allowed	Concurrent modification allowed in safe manner
Iterator is fail-fast because while concurrent modification it fails vey fast and throws Concurrent modification exception	Iterator of CHM is fail-safe. It does not throw CME while concurrent update.

Null is allowed for both key and value. Only one null key and multiple null value required,
`HashMap<Integer, String> hashMapTest=new HashMap();`
 	
`hashMapTest.put(null, "Same");`
`hashMapTest.put(1, null);`
`hashMapTest.put(3, null);`
 	
`hashMapTest.put(null, null);//second null key-No Compilation/Run time error`
`//just it will not be availabe in map`
 	
`System.out.println(hashMapTest);`
`Iterator itr=hashMapTest.entrySet().iterator();`
`while(itr.hasNext())`
`{`
`Entry entry=(Entry)itr.next();`
 	
`System.out.println(entry.getKey()+" "+entry.getValue());`
 	
`}`
Output-
{null=null, 1=null, 3=null}
null null
1 null
3 null
	Null not allowed. whether key or value
	
	`ConcurrentHashMap<Integer, String> hashMapTest=new ConcurrentHashMap();`
	`hashMapTest.put(null, "Same");`
	
	No compilation error
	Runtime exception
	Exception in thread "main" java.lang.NullPointerException
	at java.base/java.util.concurrent.ConcurrentHashMap.putVal(Unknown Source)
	at java.base/java.util.concurrent.ConcurrentHashMap.put(Unknown Source)
	at concurrentCollections.MapTest.main(MapTest.java:12)

Difference between Concurrent HashMap , SynchronizedMap and Hashtable

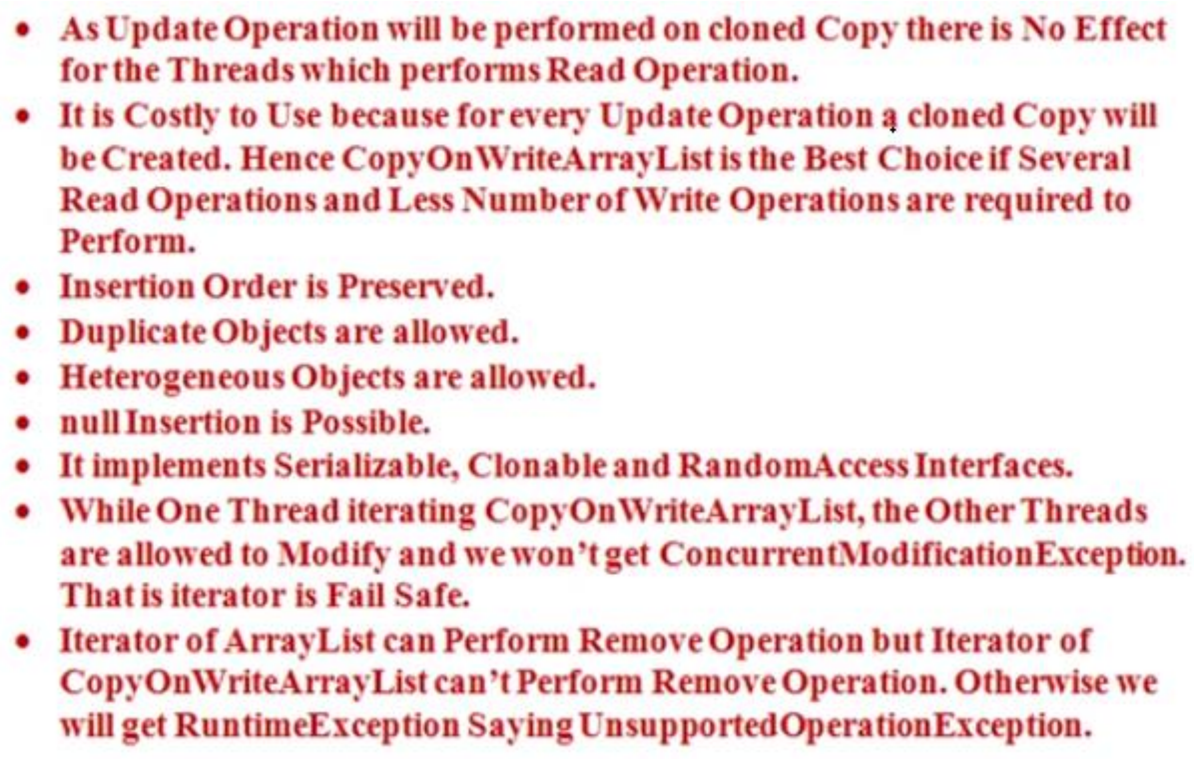
Similarity- All three are thread safe

Concurrent HashMap	SynchronizedMap	Hashtable

Thread safety without whole map object lock. Segment locking or bucket level locking works here	Thread safety with whole map object level lock	Thread safety with whole map object level lock
At a time multiple threads to read and 16 threads to write	At a time only one thread is allowed to read/write	At a time only one thread is allowed to read/write
Requires lock only for writing	Requires object level locking even for reading.	Requires object level locking even for reading
Concurrent Modification allowed.	Concurrent modification exception	Concurrent modification exception
Iterator is fail-safe	Iterator is fail-fast	Iterator is fail-fast
Null not allowed for both key and value	**One null allowed for key and multiple null values**	**Null not allowed for both key and value**

CopyOnWriteArrayList

- It is implementation class of List Interface.
- It is thread safe version/concurrent version of ArrayList.
- For every write operation a separate clone copy will be created on that cloned copy write operation will be performed so that we will get thread safety. Later these two copies will be synced by JVM automatically internally.
- Suppose we are doing 1000 of write operations then 1000 cloned copy will be created which is again a performance issue. So when we have more read operations and less write operation then CopyOnWriteArrayList is recommended.
- Functionality of CopyOnWriteArrayList is same as ArrayList except copyonwrite property.



- 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 Fail Safe.
- Iterator of ArrayList can Perform Remove Operation but Iterator of CopyOnWriteArrayList can't Perform Remove Operation. Otherwise we will get RuntimeException Saying UnsupportedOperationException.

<u>**Difference between AL and COWAL**</u>

Iterator of ArrayList is fail-first and iterator of CopyOnWriteArrayList is fail-safe.

Normal ArrayList can perform remove operation but COWAL cannot perform this operation. If trying you will get runtime UsupportedOperationException.

Constructor in COWAL

COWAL l=new COWAL();

COWAL l=new COWAL(Collection c);

COWAL l=new COWAL(Object[] a);

New methods introduced in COWAL

Boolen addIfAbsent(Object o) and addAllAbsent(Collection c)

Adds unique elements only like set. Here we can use set like mechanism based on indexing.

Add() vs addIfAbsent()

```
CopyOnWriteArrayList<String> copyOnArraylist=new CopyOnWriteArrayList();
                System.out.println("Example of COWAL");
                copyOnArraylist.addIfAbsent("One");
                copyOnArraylist.addIfAbsent("Two");
                copyOnArraylist.addIfAbsent("One");
                System.out.println(copyOnArraylist);
                System.out.println("Example of traditional ArrayList");
                ArrayList<String> arrayList=new ArrayList();
                arrayList.add("One");
                arrayList.add("Two");
                arrayList.add("One");
                System.out.println(arrayList);
```

Output-

Example of COWAL

[One, Two]

Example of traditional ArrayList

[One, Two, One]

Addall() vs addAllAbsent()

```
CopyOnWriteArrayList<String> copyOnArraylist=new CopyOnWriteArrayList();
                copyOnArraylist.addIfAbsent("One");
                copyOnArraylist.addIfAbsent("Two");
                copyOnArraylist.addIfAbsent("Three");
                ArrayList<String> arrayList=new ArrayList();
                arrayList.add("Three");
                arrayList.add("Four");
                arrayList.add("One");
                System.out.println("addAll() in ArrayList :: adda all the 
elements");
                arrayList.addAll(copyOnArraylist);
                System.out.println(arrayList);
                System.out.println("addAllAbsent in COWAL :: adds only elements 
which are not present in COWAL:: No duplicates");
```

```
		copyOnArraylist.addAllAbsent(arrayList);
		System.out.println(copyOnArraylist);
```

Output-

addAll() in ArrayList :: adda all the elements

[Three, Four, One, One, Two, Three]

addAllAbsent in COWAL :: adds only elements which are not present in COWAL:: No duplicates

[One, Two, Three, Four]

Concurrent Write Demo in COWAL

```
package copyOnWriteArrayList;

import java.util.Iterator;
import java.util.concurrent.CopyOnWriteArrayList;

public class ConcurrentUpdateInCOWAL extends Thread {
	static CopyOnWriteArrayList<String> cowal=new CopyOnWriteArrayList();
	public void run()
	{
		cowal.add("Three");
		try {
			Thread.sleep(2000);
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}

	public static void main(String[] args) {
		cowal.add("One");
		cowal.add("Two");

		ConcurrentUpdateInCOWAL t=new ConcurrentUpdateInCOWAL();
		t.start();
		cowal.add("Four");
		Iterator<String> itr=cowal.iterator();
		while(itr.hasNext())
		{
			String str=(String)itr.next();
			System.out.println(str);
		}

		System.out.println(cowal);
	}

}
```

Output-

```
One

Two

Four

Three
```

[One, Two, Four, Three]

Please note here concurrent written object is present in iterator also unlike in CHM. Because it works on indexing principle unlike buckets. Elements will be added only in forward direction following indexing principle.

Note – ArrayList iterator can remove elements but COWAL cannot remove. Throws UnsupportedOperationException.

```
cowal.add("One");
		cowal.add("Two");

		ConcurrentUpdateInCOWAL t=new ConcurrentUpdateInCOWAL();
		t.start();
		cowal.add("Four");
		Iterator<String> itr=cowal.iterator();
		while(itr.hasNext())
		{
			itr.remove();
		}
```

Output-

```
Exception in thread "main" java.lang.UnsupportedOperationException

	at java.base/java.util.concurrent.CopyOnWriteArrayList$COWIterator.remove(Unknown Source)

	at copyOnWriteArrayList.ConcurrentUpdateInCOWAL.main(ConcurrentUpdateInCOWAL.java:29)
```

If we are updating COWAL while iterating new element will not be available to iterator Since a new clone of this list has been created which contains the newly added element. But if you print list then it will be available as sync happens automatically by JVM.

Let's clarify using an example

```
CopyOnWriteArrayList<String> list=new CopyOnWriteArrayList();
		list.add("One");
		list.add("Two");
		Iterator<String> itr=list.iterator();
		list.add("Three");//While iterating we are updating
		while(itr.hasNext())
		{
			String str=(String)itr.next();
			System.out.println(str);
		}

		System.out.println("Updated List");
		System.out.println(list);
```

Output-

```
One

Two

Updated List
```

[One, Two, Three]

What will happen if we try the same with ArrayList- Runtime Exception

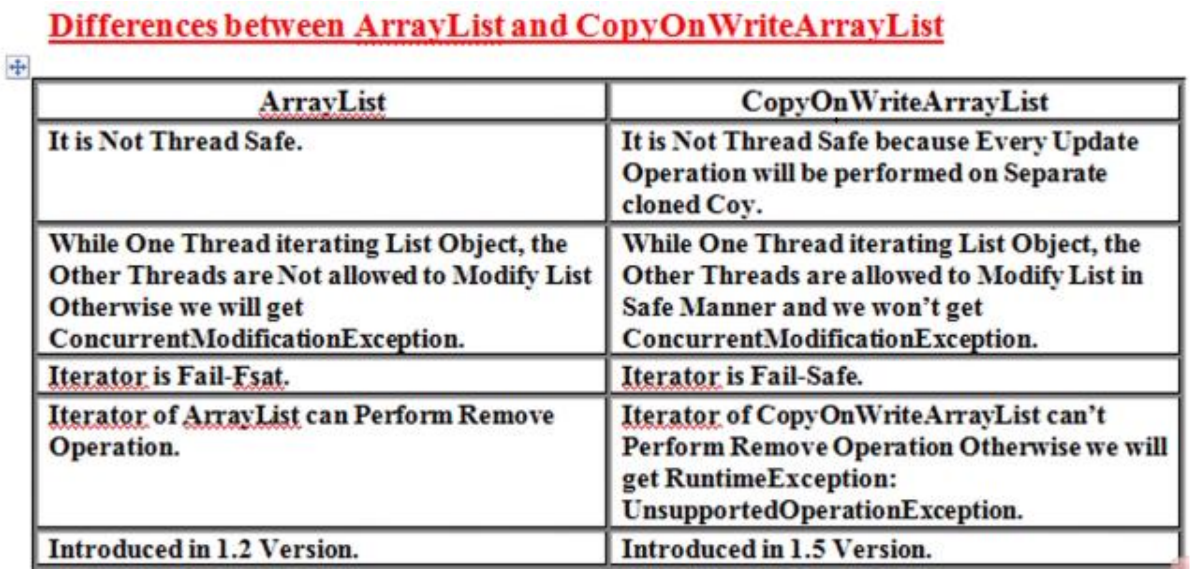
```
ArrayList<String> list=new ArrayList();
             list.add("One");
             list.add("Two");
             Iterator<String> itr=list.iterator();
             list.add("Three");//While iterating we are updating
             while(itr.hasNext())
             {
                    String str=(String)itr.next();
                    System.out.println(str);
             }
```

Error-

```
Exception in thread "main" java.util.ConcurrentModificationException
	at java.base/java.util.ArrayList$Itr.checkForComodification(Unknown Source)
	at java.base/java.util.ArrayList$Itr.next(Unknown Source)
	at copyOnWriteArrayList.ALIterator.main(ALIterator.java:16)
```

AL vs COWAL

Note: COWAL can have duplicate elements if elements are inserted using add() method. addIfAbsent () does not allow duplicates.

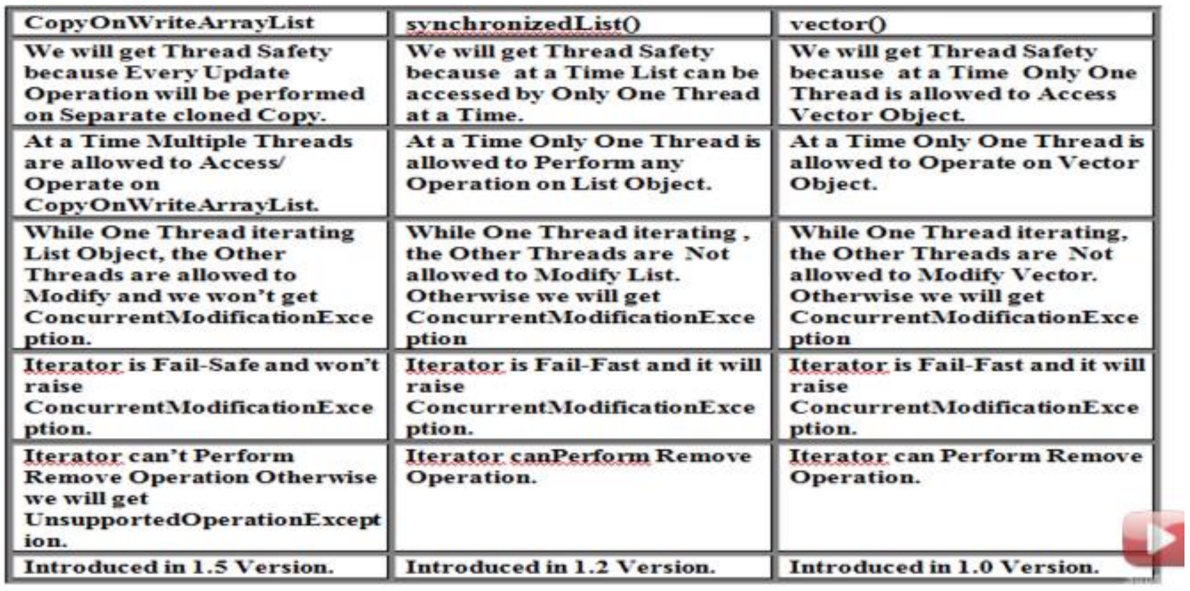


Differences between ArrayList and CopyOnWriteArrayList

ArrayList	CopyOnWriteArrayList
It is Not Thread Safe.	It is Not Thread Safe because Every Update Operation will be performed on Separate cloned Coy.
While One Thread iterating List Object, the Other Threads are Not allowed to Modify List Otherwise we will get ConcurrentModificationException.	While One Thread iterating List Object, the Other Threads are allowed to Modify List in Safe Manner and we won't get ConcurrentModificationException.
Iterator is Fail-Fsat.	Iterator is Fail-Safe.
Iterator of ArrayList can Perform Remove Operation.	Iterator of CopyOnWriteArrayList can't Perform Remove Operation Otherwise we will get RuntimeException: UnsupportedOperationException.
Introduced in 1.2 Version.	Introduced in 1.5 Version.

ArrayList vs CopyOnWriteArrayList vs Vector

Differences between CopyOnWriteArrayList, synchronizedList() and vector()

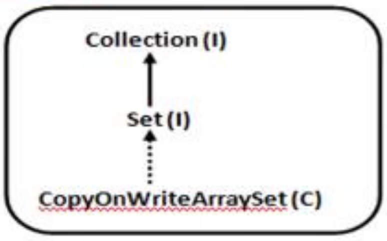


CopyOnWriteArrayList	synchronizedList()	vector()
We will get Thread Safety because Every Update Operation will be performed on Separate cloned Copy.	We will get Thread Safety because at a Time List can be accessed by Only One Thread at a Time.	We will get Thread Safety because at a Time Only One Thread is allowed to Access Vector Object.
At a Time Multiple Threads are allowed to Access/ Operate on CopyOnWriteArrayList.	At a Time Only One Thread is allowed to Perform any Operation on List Object.	At a Time Only One Thread is allowed to Operate on Vector Object.
While One Thread iterating List Object, the Other Threads are allowed to Modify and we won't get ConcurrentModificationException.	While One Thread iterating , the Other Threads are Not allowed to Modify List. Otherwise we will get ConcurrentModificationException	While One Thread iterating, the Other Threads are Not allowed to Modify Vector. Otherwise we will get ConcurrentModificationException
Iterator is Fail-Safe and won't raise ConcurrentModificationException.	Iterator is Fail-Fast and it will raise ConcurrentModificationException.	Iterator is Fail-Fast and it will raise ConcurrentModificationException.
Iterator can't Perform Remove Operation Otherwise we will get UnsupportedOperationException.	Iterator canPerform Remove Operation.	Iterator can Perform Remove Operation.
Introduced in 1.5 Version.	Introduced in 1.2 Version.	Introduced in 1.0 Version.

CopyOnWriteSet

- Internal functionality is same as COWAL.
- Null allowed
- Duplicate not allowed.
- Insertion order maintained as internally it implements COWAL

CopyOnWriteArraySet :



- It is a Thread Safe Version of Set.
- Internally Implement by CopyOnWriteArrayList.
- Insertion Order is Preserved.
- Duplicate Objects are Notallowed.
- 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 PerformOnly Read Operation and won't Perform Remove Operation. Otherwise we will get RuntimeException: UnsupportedOperatonException.

Constructors of COWAS

COWAS s=new COWAS();

COWAS s=new COWAS(Collection c)

There are no new methods in COWAS. As we know SET interface does not contains any methods so what ever methods are present in Collection Interface are present in COWAS.

Program-

```
package copyOnWriteArraySet;

import java.util.Iterator;
import java.util.concurrent.CopyOnWriteArraySet;

public class COWASDemo {

	public static void main(String[] args) {
		CopyOnWriteArraySet<String> copySet=new CopyOnWriteArraySet();
		copySet.add("One");
		copySet.add("Two");
		copySet.add(null);
		copySet.add("Two");//duplicate
		copySet.add(null);//second null element--considered as duplicate
		Iterator<String> itr=copySet.iterator();
		copySet.add("Thee");//adding when iterator is iterating will not be
available to
		//iterator as it maintains insertion order.
		//Three is present is clone Set Object
		while(itr.hasNext())
		{
			String str=(String)itr.next();
			System.out.println(str);
		}
		System.out.println("----------------------------");
		System.out.println(copySet);//Clone set is synced here


	}

}
```

Output-

```
One

Two

null

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

[One, Two, null, Thee]
```