Concurrent Collections

1. Need for Concurrent Collections

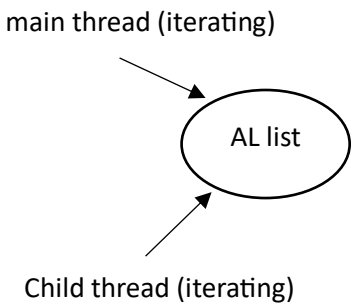
a) Traditional Collection objects (like ArrayList, HashMap etc.) can be accessed by Multiple Threads simultaneously & there may be a chance of Data Inconsistency problems & hence these are not Thread safe.

b) Already existing thread safe collections (like Vector, Hashtable, synchronizedList (), synchronizedSet (), synchronizedMap ()) are not up to mark performance wise because for any operation even for read operation, only one thread is allowed to operate on the same object others have to wait.

c) While one thread is iterating collection, the other threads are not allowed to modify Collection object simultaneously if we're trying to modify then we will get **RE: ConcurrentModificationException**.

```
class MyThread extends Thread {
   public static ArrayList list = new ArrayList();
   public void run () {
      try { Thread.sleep(2000); } catch(InterruptedException ie){}
      System.out.println("Child Thread updating list");
      list.add("D");
   }

   public static void main (String [] args) throws InterruptedException {
      list.add("A");   list.add("B");     list.add("C");
      MyThread t = new MyThread ();
      t.start();
      Iterator itr = l.iterator();
      while(itr.hasNext()) {
         String s1 = (String)itr.next();
         System.out.println("Main thread iterating list & current object is: " + s1);
         Thread.sleep(3000);
      }
      System.out.println(l);
   }
}
```



Output:

```
Main thread iterating list & current object is: A
Child Thread updating list
Exception in thread "main" java.util.ConcurrentModificationException
```

To overcome these problems, SUN people introduced Concurrent Collections in Java 1.5 version. (Traditional Collection objects are not suitable for scalable Multithreaded applications)

Diff b/w Traditional & Concurrent Collection

No.	Traditional Collection	Concurrent Collection
1.	Traditional Collection may or may not be thread safe.	Concurrent Collection are always Thread – safe.
2.	Low Performance because of synchronization & waiting of threads.	Performance of Concurrent Collections is more because of different locking mechanism.
3.	While one thread interacting Collection, the other threads are not allowed to modify the collection. It will throw Runtime Exception: ConcurrentModificationException.	While one thread interacting Collection, the other threads are allowed to modify Collection in safe manner. Hence, Concurrent collections never threw ConcurrentModificationException.

2. The important concurrent classes are

a) ConcurrentHashMap
b) CopyOnWriteArrayList
c) CopyOnWriteArraySet

ConcurrentMap interface

```
public interface ConcurrentMap<K, V> extends Map<K, V>
```

- It is present in java.util.Concurrent package.
- It contains all Map(I) methods as well as 3 specific methods:
  1. Object putIfAbsent (Object key, Object value)
  2. boolean remove (Object key, Object value)
  3. boolean replace (Object key, Object oldValue, Object newValue)

1. Object putIfAbsent (Object key, Object value)

- to add Entry to the Map if the specified key is not already available.

```
Object putIfAbsent (Object key, Object value) {
    if(!map.containsKey(key))
        map.put(key, value);
    else
        map.get(key);
}
```

put ()	putIfAbsent ()
If the key is already available, oldValue will be replaced with newValue & returns oldValue.	If the key is already present then Entry won't be added & returns old associated value. If key is not available then only entry will be added.

```
import java.util.concurrent.ConcurrentHashMap;

class ConcurrentDemo {
  public static void main (String [] args) {
     ConcurrentHashMap m = new ConcurrentHashMap ();
     m.put (101, "Sam"); m.put (101, "Elena");
     System.out.println(m);
     m.putIfAbsent (101, "Will");
     System.out.println(m);
  }
}
```

Output:

```
{101=Elena}
{101=Elena}
```

2. boolean remove (Object key, Object value)

- removes the entry if the key associated with specified value only.

```
boolean remove (Object key, Object value) {
    if(!map.containsKey(key) && map.get(key).equals(value)) {
       map.remove (key);
       return true;
    }
    return false;
}
```

```
import java.util.concurrent.ConcurrentHashMap;

class ConcurrentDemo {
  public static void main (String [] args) {
     ConcurrentHashMap m = new ConcurrentHashMap ();
     m.put (101, "Sam");
     m.remove (101, "Elena");
     System.out.println(m);
     m.remove (101, "Sam");
     System.out.println(m);
  }
}
```

Output:

```
{101=Sam}
{}
```

3. boolean replace (Object key, Object oldValue, Object newValue)

- replaces oldValue with newValue if the key – value is matched.

```
boolean replace (Object key, Object oldValue, Object newValue) {
    if(!map.containsKey(key) && map.get(key).equals(oldValue)) {
       map.put(key, newValue);
       return true;
    }
    return false;
}
```

```
import java.util.concurrent.ConcurrentHashMap;

class ConcurrentDemo {
   public static void main (String [] args) {
     ConcurrentHashMap m = new ConcurrentHashMap ();
     m.put (101, "Sam");
     m.replace (101, "Elena", "Will");
     System.out.println(m);
     m.replace (101, "Sam", "Sony");
     System.out.println(m);
  }
}
```

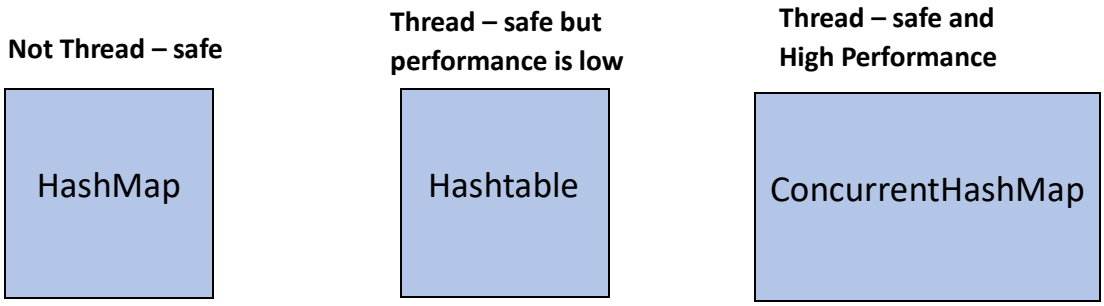
Output:

```
{101=Sam}
{101=Sony}
```

a) ConcurrentHashMap

```
public class ConcurrentHashMap<K, V> extends AbstractMap<K, V>
                                     implements ConcurrentMap<K, V>, Map<K, V>
```

- The underlying Data structure is Hashtable.
- ConcurrentHashMap allows concurrent read & thread safe update operations.
- To perform read operation, Thread won't require any lock. But to perform update operation, Thread will require lock but it's the lock of only a particular part of Map (Bucket level lock / Segment lock) instead of total Map.
- Instead of whole Map, Concurrent update achieved by internally diving Map into smaller portion which is defined by Concurrency level.
- The default concurrency level is 16 (one lock for each bucket) i.e., ConcurrentHashMap allows any no. of simultaneous read operation & simultaneously 16 write/update operations.
- Concurrency level can be different like 8 (1 lock per 2 buckets), 32 (2 locks per 1 bucket) etc.
- null is not allowed for both key or values.
- Concurrent Collections are best suitable for Multithreaded Scalable application.
- While one thread is iterating, the other thread can perform update operation & Concurrent HashMap never throw **ConcurrentModificationException**.



Constructors

1. ConcurrentHashMap m = new ConcurrentHashMap () – creates an empty ConcurrentHashMap object with initial capacity as 16, default fillRatio as 0.75, & default concurrency level as 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);

```
class MyThread extends Thread {
   static ConcurrentHashMap m = new ConcurrentHashMap ();
   public void run () {
      try { Thread.sleep(2000); } catch(InterruptedException ie) {}
      System.out.println("Child Thread updating Map");
      m.put(103, "C");
      m.put(104, "D");
   }

   public static void main (String [] args) throws InterruptedException {
      m.put(101, "A"); m.put(102, "B");
      System.out.println(m);
      MyThread t = new MyThread ();
      t.start();
      Set s1 = m.keySet();
      Iterator itr = s1.iterator();
      while (itr.hasNext()) {
         Integer i1 = (Integer)itr.next();
         System.out.println("main thread iterating Map &" +
          " Current entry is: " + i1 + "...." + m.get(i1));
         Thread.sleep(3000);
      }
      System.out.println(m);
   }
}
```

Output:

```
{101=A, 102=B}
main thread iterating Map & Current entry is: 101....A
Child Thread updating Map
main thread iterating Map & Current entry is: 102....B
main thread iterating Map & Current entry is: 103....C
main thread iterating Map & Current entry is: 104....D
{101=A, 102=B, 103=C, 104=D}
```

Note: While one thread is iterating, the other thread is allowed to perform any modification to the Map but that updation is available to the iterator or not, there is no guarantee.

No.	HashMap	ConcurrentHashMap
1.	It's not thread – safe	It's thread – safe
2.	Relatively performance is high because threads are not required to wait.	Relatively performance is low because sometimes threads are required to wait to operate on ConcurrentHashMap.
3.	While one thread is iterating HashMap, the other threads are not allowed to modify Map object otherwise we will get RE: ConcurrentModificationException.	While one thread is iterating ConcurrentHashMap, the other threads are allowed to modify Map objects in safe manner & it won't throw ConcurrentModificationException.
4.	Iterator of HashMap is Fail – fast & it throws ConcurrentModificationException.	Iterators of ConcurrentHashMap is Fail – safe & it won't throw ConcurrentModificationException.
5.	Null value is allowed for both key & value.	Null is not allowed for both otherwise RE: NullPointerException
6.	Introduced in 1.2 version	Introduced in 1.5 version

No.	ConcurrentHashMap	synchronizedMap () / Hashtable
1.	We will get Thread safety without locking total Map object just with bucket level lock.	We will get Thread safety by locking whole Map object.
2.	At a time, Multiple threads are allowed to operate on Map object in safe manner.	At a time, only thread is allowed to perform any operation on Map object.
3.	Read operation can be performed without lock but write operation can be performed with Bucket level lock.	Every read & write operations require total Map object lock.
4.	While one thread iterating Map object, the other threads are allowed to modify Map & we won't get ConcurrentModificationException.	While one thread iterating Map object, the other threads are not allowed to modify Map otherwise we will get RE: ConcurrentModificationException.
5.	Iterator of ConcurrentHashMap is Fail – safe & won't raise ConcurrentModificationException.	Iterator of synchronizedMap () & Hashtable are Fail – Fast & they will raise ConcurrentModificationException.

b) CopyOnWriteArrayList (COWAL)

public class CopyOnWriteArrayList<K, V> **extends** Object
implements List<E>, RandomAccess, Cloneable, Serializable

- It is a thread safe version of ArrayList as the name suggests CopyOnWriteArrayList creates a cloned copy of underlying ArrayList for every update operation at certain point. Both will be synchronized automatically which is taken care by JVM.
- 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 & less no. of write operations are required to perform.
- Insertion order is preserved. Duplicate objects are allowed. Heterogenous objects are allowed. Null insertion is possible.
- While one thread is iterating CopyOnWriteArrayList, the other threads are allowed to modify & we won't get ConcurrentModificationException. This iteration is called **Fail Safe**. Iterator of ArrayList can perform remove operation but iterator of CopyOnWriteArrayList can't perform remove operation otherwise we will get RE: UnsupportedOperationException.

Constructors

1. CopyOnWriteArrayList l = new CopyOnWriteArrayList ();
2. CopyOnWriteArrayList l = new CopyOnWriteArrayList (Collection c);
3. CopyOnWriteArrayList l = new CopyOnWriteArrayList (Object [] o);

Methods

- All methods present in Collection i/f & List i/f can be applied in COWAL.
- It has 2 specific methods:
  1. boolean **addIfAbsent** (Object o)
  2. int **addAllAbsent** (Collection c)

No.	ArrayList	CopyOnWriteArrayList
1.	It is not thread – safe.	It is thread – safe because every update operation will be performed on separate cloned copy.
2.	While one thread is iterating list, the other threads are not allowed to modify list, otherwise we will get ConcurrentModificationException.	While one thread is iterating list object, the other threads are allowed to modify list in safe manner & we won't get ConcurrentModificationException.
3.	Iterator is Fail – Fast	Iterator is Fail – safe
4.	Iterator of AL can perform both read & remove operations.	Iterator of COWAL can perform only read operation but not remove operation otherwise we will get RE: UnsupportedOperationException.
5.	Present in java.util package & introduced in 1.2 version	Present in java.util package & introduced in 1.5 version.

No.	CopyOnWriteArrayList	synchronizedList () / Vector
1.	We will get thread safety because every update operation will be performed on separated cloned copy.	We will get thread safety because at a time, list can be accessed by only one thread.
2.	At a time, multiple threads are allowed to operate on COWAL object.	At a time only one thread is allowed to operate on list object / Vector object.
3.	Iterator is fail – safe & won't raise ConcurrentModificationException.	Iterator is Fail – fast & will raise ConcurrentModificationException.
4.	While one thread iterating list object, the other threads are allowed to modify & we won't get ConcurrentModificationException.	While one thread is iterating List Object / Vector object, the other threads are not allowed to modify otherwise we will get CME.
5.	Iterator can perform only read but not remove operation otherwise we will get RE: UnsupportedOperationException.	Iterator can perform both read & remove operations.

c) CopyOnWriteArraySet (COWAS)

- It is thread safe version of set. Internally implemented 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's not recommended to use COWAS.
- While one thread iterating Set, the other threads are allowed to modify set & we won't get CME i.e., iterator is fail – safe.
- Iterator of COWAS can perform only read operation & won't perform remove operation otherwise we will get RE: UnsupportedOperationException.

Constructors

1. CopyOnWriteArraySet s = new CopyOnWriteArraySet ();
2. CopyOnWriteArraySet s = new CopyOnWriteArraySet (Collection c);

Methods – No new methods as Duplicate objects are not allowed. All methods from Collection & Set are valid.

No.	CopyOnWriteArraySet	synchronizedSet ()
1.	It is thread safe because every update operation will be performed on separate cloned copy.	It is thread safe because at a time, only one thread is allowed to operate.
2.	While one thread is iterating Set, other threads are allowed to modify & we won't get CME.	While one thread is iterating, the other threads are allowed to modify Set otherwise we will get CME.
3.	Iterator is fail – safe	Iterator is fail – fast
4.	Iterator can perform only read operation but not remove operation otherwise we will get RE: UnsupportedOperationException.	Iterator can perform both read & remove operations.