# **Concurrent Collections in Java**

Concurrent collection classes are used to ensure safe and concurrent access to data by multiple threads (which is not the case in normal collection classes like list, set, map etc).The concurrent collection classes that we would be talking about in this chapter are:

1. CopyOnWriteArrayList
2. CopyOnWriteArraySet
3. ConcurrentHashmap

Concurrent collection classes are meant for synchronized access to threads. Although, we already have some classes like Vector and Hashtable, which are synchronized or thread-safe, but they have some cons associated with them, and they are not the best solutions. Therefore, a new types of classes have been introduced as part of concurrent collections in java which serve this purpose.

These classes are part of java.util.concurrent package. Let’s first understand the need of concurrent collection classes.

## **Need of concurrent collections**

1. Most of the classes in collections framework like Arraylist, LinkedList, HashSet, HashMap, LinkedHashMap etc are not thread safe. If we use these collections in a multithreaded environment, then we will not get the right results. This raises the requirement to have thread safe classes.
2. Although, we can make make a list, or set synchronized, for that matter, using methods like Collections.synchronizedList(list), but it is not an efficient way.
3. If we use Collections.synchronizedList() method, then the entire list gets locked by one thread. Even if all threads are readers threads, then also, the they can access the list in a synchronized way. This increases the response time significantly.

## **Concurrent collection classes in java**

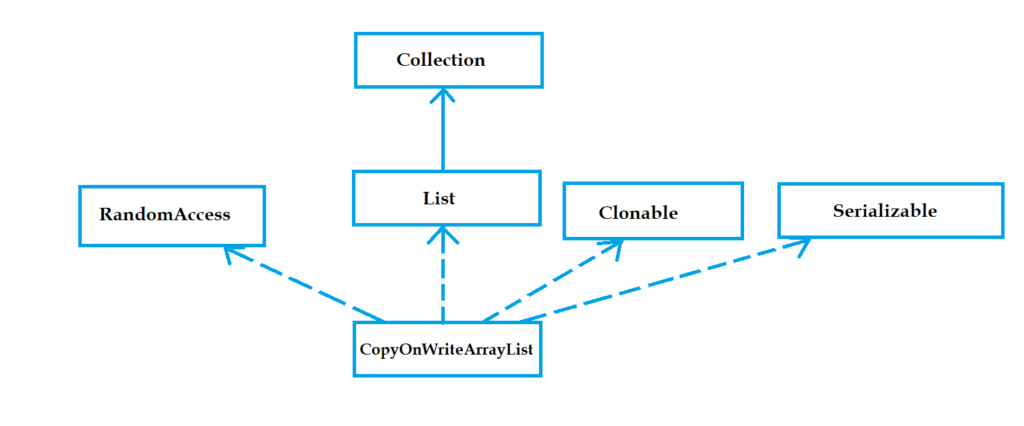
1. CopyOnWriteArrayList
2. CopyOnWriteArraySet
3. ConcurrentHashmap

## **CopyOnWriteArrayList**

CopyOnWriteArrayList is part of collections framework with some differences with ArrayList. CopyOnWriteArrayList is thread safe and can be used in multithreaded environment. Let’s have a look at the features of CopyOnWriteArrayList

1. The CopyOnWriteArrayList is a thread safe version of ArrayList. If we are making modifications like adding, removing elements in CopyOnWriteArrayList, then JVM does so by creating a new copy of it by the use of Cloning.
2. CopyOnWriteArrayList is costly if used in case of more update operations. Because when changes are made, JVM has to create a cloned copy of the underlying array and add/update elements to it.
3. Multiple threads can read the data from CopyOnWriteArrayList, but only one thread can write data at a particular time.
4. We can add duplicate elements in it.
5. CopyOnWriteArrayList is the best choice in multithreading, if there are more read operations.

## **Class hierarchy**

****

## **Constructor Summary**

CopyOnWriteArrayList()

CopyOnWriteArrayList(Collection<? extends E> c)

CopyOnWriteArrayList(E[] toCopyIn)

## **CopyOnWriteArrayList Example**

package com.javatrainingschool; import java.util.Arrays; import java.util.Iterator; import java.util.List; import java.util.concurrent.CopyOnWriteArrayList; public class CopyOnWriteArrayListExample { public static void main(String[] args) { List<String> list = Arrays.asList("CV Raman", "Homi Bhabha", "Ramanujan"); CopyOnWriteArrayList<String> cowArrayList = new CopyOnWriteArrayList<String>(list); System.out.println("List = " + cowArrayList); Iterator<String> iterator1 = cowArrayList.iterator(); // adding another element cowArrayList.add("Vikram Sarabhai"); while(iterator1.hasNext()) { System.out.println("Element from iterator1 : " + iterator1.next()); } Iterator<String> iterator2 = cowArrayList.iterator(); while(iterator2.hasNext()) { System.out.println("Element from iterator2 : " + iterator2.next()); } } }

Output :

List = [CV Raman, Homi Bhabha, Ramanujan]

Element from iterator1 : CV Raman

Element from iterator1 : Homi Bhabha

Element from iterator1 : Ramanujan

Element from iterator2 : CV Raman

Element from iterator2 : Homi Bhabha

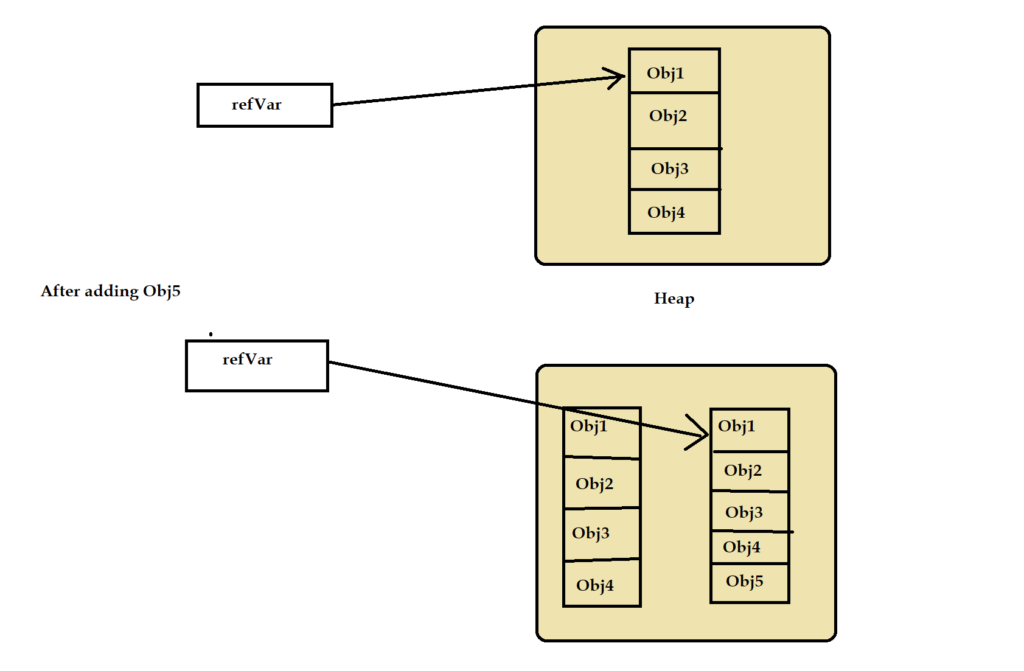
Element from iterator2 : Ramanujan

Element from iterator2 : Vikram Sarabhai

## **How CopyOnWriteArrayList work internally?**

CopyOnWriteArrayList class does every write operation (add, set, remove, etc) in a new copy of the list. JVM creates a new copy of the list and modifications are done in that. Using this technique, thread-safety is achieved without a need for synchronization.

Note -> Any number of reader threads can access CopyOnWriteArrayList simultaneously. And reader and writer threads do not block each other.

CopyOnWriteArrayList internal working

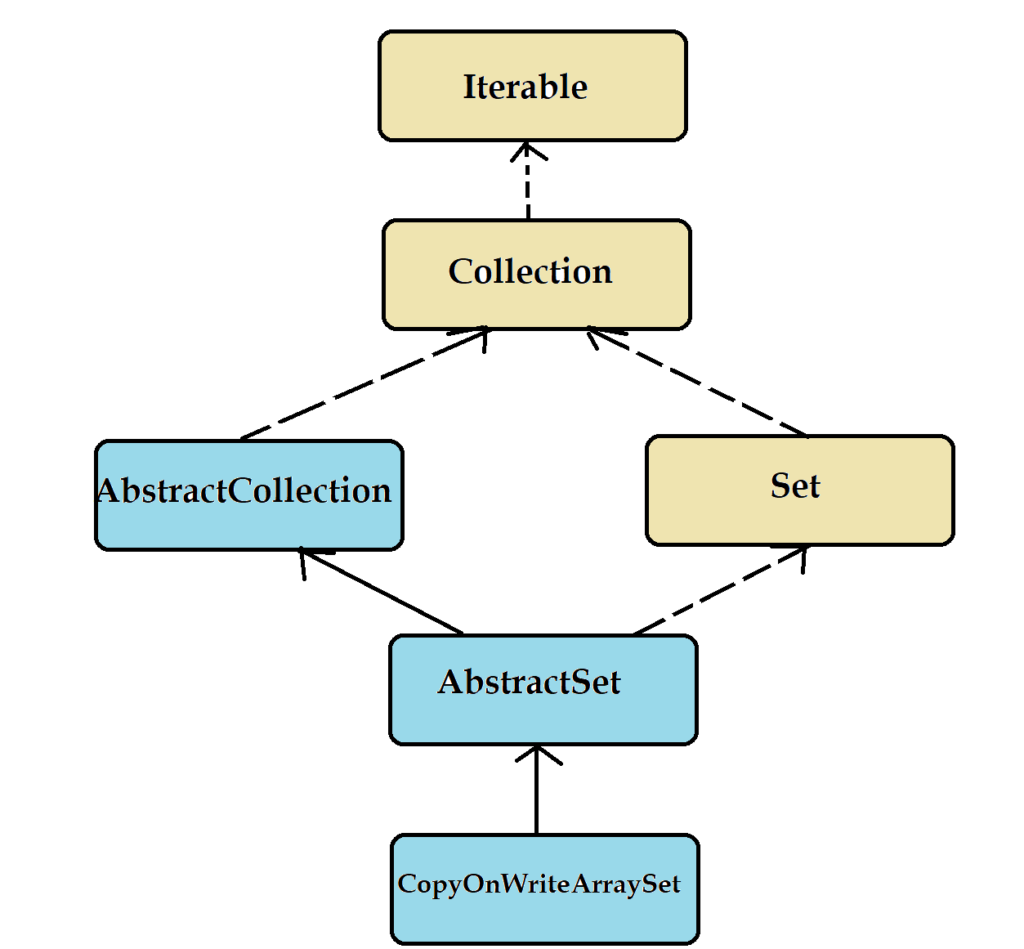
When we call iterator() or listIterator() methods on CopyOnWriteArrayList, it returns an iterator object that holds immutable snapshot of the list objects. After iterator is created, if any other thread makes changes in the list, then that will not reflect in iterator objects. Due to which, we can iterate over the list in a safe way, without bothering about the concurrent modification.

CopyOnWriteArrayList is useful in following cases:

1. When we need to use a list in a multithreaded environment, then multiple threads can perform read and write operations simultaneously.
2. When read operations are more and write operations are less. If it is vice versa, then we can use Vector or synchronized list.

## **CopyOnWriteArraySet**

CopyOnWriteArraySet is like CopyOnWriteArrayList. It is thread safe version of HashSet.



## **Class declaration**

class CopyOnWriteArraySet<E> extends AbstractSet<E> implements java.io.Serializable

## **Constructor summary**

CopyOnWriteArraySet()

CopyOnWriteArraySet(Collection<? extends E> c)

## **Example**

package com.javatrainingschool; import java.util.Arrays; import java.util.Iterator; import java.util.List; import java.util.concurrent.CopyOnWriteArraySet; public class CopyOnWriteArrayListExample { public static void main(String[] args) { List<String> list = Arrays.asList("CV Raman", "Homi Bhabha", "Ramanujan", "Ramanujan", "Homi Bhabha"); CopyOnWriteArraySet<String> cowArraySet = new CopyOnWriteArraySet<String>(list); System.out.println("Set = " + cowArraySet); Iterator<String> iterator1 = cowArraySet.iterator(); // adding another element cowArraySet.add("Vikram Sarabhai"); while(iterator1.hasNext()) { System.out.println("Element from iterator1 : " + iterator1.next()); } Iterator<String> iterator2 = cowArraySet.iterator(); while(iterator2.hasNext()) { System.out.println("Element from iterator2 : " + iterator2.next()); } } }

Output :

Set = [CV Raman, Homi Bhabha, Ramanujan]

Element from iterator1 : CV Raman

Element from iterator1 : Homi Bhabha

Element from iterator1 : Ramanujan

Element from iterator2 : CV Raman

Element from iterator2 : Homi Bhabha

Element from iterator2 : Ramanujan

Element from iterator2 : Vikram Sarabhai

## **ConcurrentHashMap**

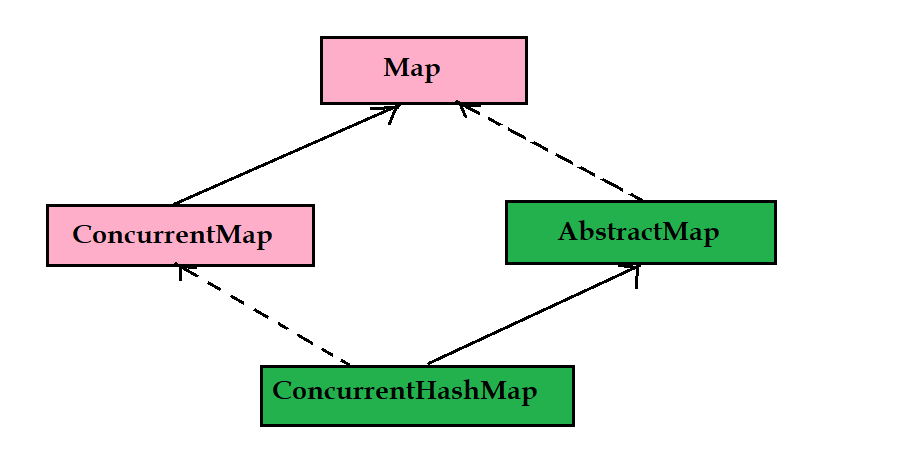
CosurrentHashMap, as its name suggests, provides concurrent access to multiple threads. It is thread safe version of HashMap. We know, HashMap is not synchronized and therefore is not thread safe. To overcome this issue, collections framework provides ConcurrentHashMap.

ConcurrentHashMap provides similar functionalities like HashMap. The only difference is that it internally maintains concurrency.  
It is concurrent version of HashMap. It internally maintains a HashTable, which is divided into segments. The number of segments depend on the level of concurrency specified in the ConcurrentHashMap. By default, it divides it into 16 segments. It doesn’t lock the entire map unlike HashTable or Synchronized HashMap. It only locks the particular segment of HashMap where write operation is going on.

## **Class declaration**

class ConcurrentHashMap<K,V> extends AbstractMap<K,V> implements ConcurrentMap<K,V>, Serializable

## **Class hierarchy**

****

## **Constructors summary**

ConcurrentHashMap() //default constructor

ConcurrentHashMap(Map<? extends K, ? extends V> m)

ConcurrentHashMap(int initialCapacity, float loadFactor)

ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel)

## **Features of ConcurrentHashMap**

1. ConcurrentHashMap internally uses HashTable as its data structure. It is thread safe just like Hashtable. It provides all the functionalities of HashMap except thread safety.
2. ConcurrentHashMap internally divides it into segments. Each segment works independently and can be accessed by different reader threads simultaneously. However, each segment can be accessed only by one writer thread at a time. This also means, a concurrent hash map can be accessed by as many writer threads together as there are segments.
3. The default level of concurrency is 16. Which means by default, there are 16 segments.
4. Read operations don’t require locking of cocurrent hash map, where as write operations do require locking.
5. Locking is known as segment or bucket locking.
6. Concurrent hash map doesn’t allow null key or null values.

## **ConcurrentHashmap code example**

package com.javatrainingschool; import java.util.concurrent.ConcurrentHashMap; public class ConcurrentHashMapExample { public static void main(String[] args) { ConcurrentHashMap<Integer, String> cMap = new ConcurrentHashMap<>(); cMap.put(1, "Taj Mahal"); cMap.put(2, "Qutab Minar"); cMap.put(3, "Char Minar"); for(Integer key : cMap.keySet()) { System.out.println(key + " : " + cMap.get(key)); } } }

Output :

1 : Taj Mahal

2 : Qutab Minar

3 : Char Minar

## **Concurrent modification in ConcurrentHashMap**

ConcurrentHashMap returns a fail safe iterator. It means, using this iterator, we can modify ConcurrentHashMap while iterating it. Let’s see one example

package com.javatrainingschool; import java.util.Iterator; import java.util.concurrent.ConcurrentHashMap; public class ConcurrentHashMapExample { public static void main(String[] args) { ConcurrentHashMap<Integer, String> cMap = new ConcurrentHashMap<>(); cMap.put(1, "Taj Mahal"); cMap.put(2, "Qutab Minar"); cMap.put(3, "Char Minar"); Iterator<Integer> it = cMap.keySet().iterator(); while(it.hasNext()) { int key = it.next(); if(key == 2) cMap.put(2, "Gateway of India"); System.out.println(key + " : " + cMap.get(key)); } } }

Output :

1 : Taj Mahal

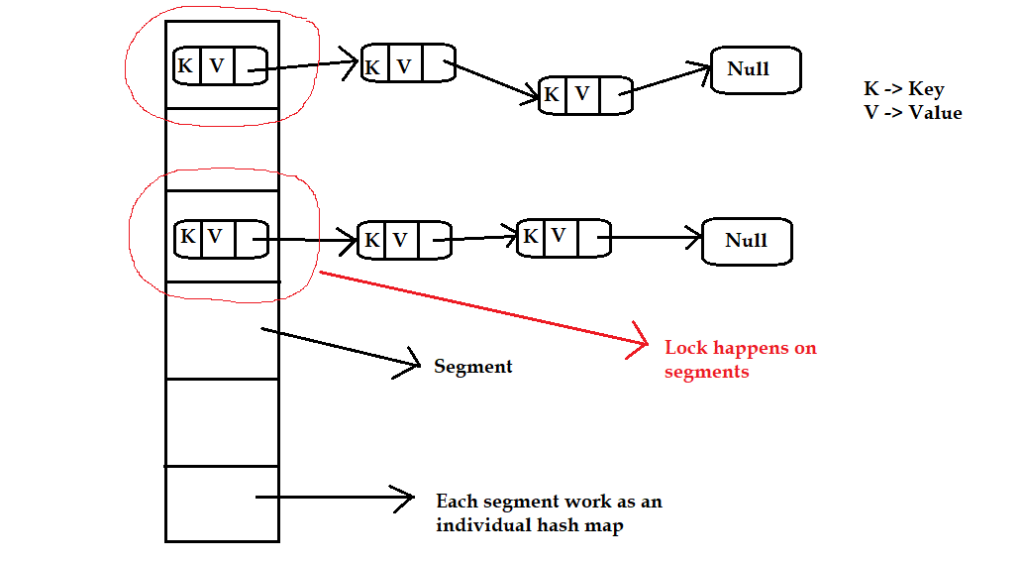
2 : Gateway of India

3 : Char Minar

## **Fail safe iterator returned by ConcurrentHashMap**

In the above example, we saw that we can modify ConcurrentHashMap while iterating it. It became possible because the iterator returned by ConcurrentHashMap is fail safe. Which means we can modify it while iterating.

## **Internal working of ConcurrentHashMap**

****

ConcurrentHashMap performs segment locking in a multithreading environment. It allows performing read operations concurrently without any blocking of threads.  
It performs retrieval operations without blocking but write operations may conflict each other if they are working on the same segment. Each segment works as an individual HashMap. So, if one thread is writing onto one segment, it locks that segment. Now until that thread is done, no other thread will get access to that segment. However, reader threads can still access this segement. If another thread, wants to write to another segment at the same time, it is free to do so. It will acquire lock on that segment and will start doing its job.  
Read operations return most recently updated operations, which means the it may not fetch the currently updating value.  
Memory visibility for the read operations is ensured by volatile reads.