**Collections Interview Questions**

Contents

[1. What is ConcurrentHashMap? 2](#_Toc483898221)

[Summary 5](#_Toc483898222)

[2. Java Concurrent Collection - CopyOnWriteArrayList Examples? 6](#_Toc483898223)

[1. Why CopyOnWriteArrayList? 6](#_Toc483898224)

1. **What is ConcurrentHashMap?**

ConcurrentHashMap in Java is introduced as an alternative of Hashtable in Java 1.5 as part of Java concurrency package.

Prior to Java 1.5 if you need a Map implementation, which can be safely used in a concurrent and multi-threaded Java program, then, you only have Hashtable or synchronizedMap because HashMap is not thread-safe.

With ConcurrentHashMap, now you have a better choice; because not only it can be safely used in the concurrent multi-threaded environment but also provides better performance over Hashtable and synchronizedMap

ConcurrentHashMap performs better than earlier two because it only locks a portion of Map, instead of whole Map, which is the case with Hashtable and synchronized Map.

CHM allows concurred read operations and the same time maintains integrity by synchronizing write operations. Since update operations like put(), remove(), putAll() or clear() is not synchronized, concurrent retrieval may not reflect most recent change on Map.

In case of putAll() or clear(), which operates on whole Map, concurrent read may reflect insertion and removal of only some entries.

Another important point to remember is iteration over CHM, Iterator returned by keySet of ConcurrentHashMap are weekly consistent and they only reflect state of ConcurrentHashMap and certain point and may not reflect any recent change.

Iterator of ConcurrentHashMap's keySet area also fail-safe and doesn’t throw ConcurrentModificationExceptoin..

|  |
| --- |
| //Initialize ConcurrentHashMap instance  ConcurrentHashMap<String, Integer> m = new ConcurrentHashMap<String, Integer>();    //Print all values stored in ConcurrentHashMap instance  for each (Entry<String, Integer> e : m.entrySet())  {  system.out.println(e.getKey()+"="+e.getValue());  } |

Above code is reasonably valid in multi-threaded environment in your application.

The reason, I am saying “reasonably valid” is that, above code yet provides thread safety, still it can decrease the performance of application. And ConcurrentHashMap was introduced to improve the performance while ensuring thread safety, right??

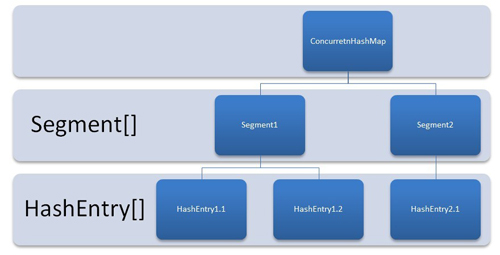
So, what is that we are missing here??

To understand that we need to understand the internal working of ConcurrentHashMap class. And the best way to start is look at the constructor arguments.

Fully parametrized constructor of ConcurrentHashMap takes 3 parameters, initialCapacity, loadFactor and concurrencyLevel.

* initialCapacity
* loadFactor
* concurrencyLevel

First two are fairly simple as their name implies but last one is tricky part. This denotes the number of shards. It is used to divide the ConcurrentHashMap internally into this number of partitions and equal number of threads are created to maintain thread safety maintained at shard level.



The default value of “concurrencyLevel” is 16. It means 16 shards whenever we create an instance of ConcurrentHashMap using default constructor, before even adding first key-value pair.

It also means the creation of instances for various inner classes like ConcurrentHashMap$Segment, ConcurrentHashMap$HashEntry[] and ReentrantLock$NonfairSync.

In most cases in normal application, a single shard is able to handle multiple threads with reasonable count of key-value pairs. And performance will be also optimal. Having multiple shards just makes the things complex internally and introduces a lot of un-necessary objects for garbage collection, and all this for no performance improvement.

The extra objects created per concurrent hashmap using default constructor are normally in ratio of 1 to 50 i.e. for 100 such instance of ConcurrentHashMap, there will be 5000 extra objects created.

Based on above, I will suggest to use the constructor parameters wisely to reduce the number of unnecessary objects and improving the performance.

A good approach can be having initialization like this:

ConcurrentHashMap<String, Integer> instance = new ConcurrentHashMap<String, Integer>(16, 0.9f, 1);

An initial capacity of 16 ensures a reasonably good number of elements before resizing happens. Load factor of 0.9 ensures a dense packaging inside ConcurrentHashMap which will optimize memory use. And concurrencyLevel set to 1 will ensure that only one shard is created and maintained.

Please note that if you are working on very high concurrent application with very high frequency of updates in ConcurrentHashMap, you should consider increasing the concurrencyLevel more than 1, but again it should be a well calculated number to get the best results.

## **Summary**

* + - ConcurrentHashMap allows concurrent read and thread-safe update operation.
    - During the update operation, ConcurrentHashMap only locks a portion of Map instead of whole Map.
    - The concurrent update is achieved by internally dividing Map into the small portion which is defined by concurrency level.
    - Choose concurrency level carefully as a significantly higher number can be a waste of time and space and the lower number may introduce thread contention in case writers over number concurrency level.
    - All operations of ConcurrentHashMap are thread-safe.
    - Since ConcurrentHashMap implementation doesn't lock whole Map, there is chance of read overlapping with update operations like put() and remove(). In that case result returned by get() method will reflect most recently completed operation from there start.
    - Iterator returned by ConcurrentHashMap is weekly consistent, fail-safe and never throw ConcurrentModificationException. In Java.
    - ConcurrentHashMap doesn't allow null as key or value.
    - You can use ConcurrentHashMap in place of Hashtable but with caution as CHM doesn't lock whole Map.
    - During putAll() and clear() operations, the concurrent read may only reflect insertion or deletion of some entries.

1. **Java Concurrent Collection - CopyOnWriteArrayList Examples?**

## **1. Why CopyOnWriteArrayList?**

Basically, a CopyOnWriteArrayList is similar to an ArrayList, with some additional and more advanced thread-safe features.

You know, ArrayList is not thread-safe so it’s not safe to use in multi-threaded applications. We can achieve thread-safe feature for an ArrayList by using a synchronized wrapper like this:

|  |
| --- |
| List<String> unsafeList = new ArrayList<>();  List<String> safeList = Collections.synchronizedList(unsafeList);    safeList.add("Boom");   // safe to use with multiple threads |