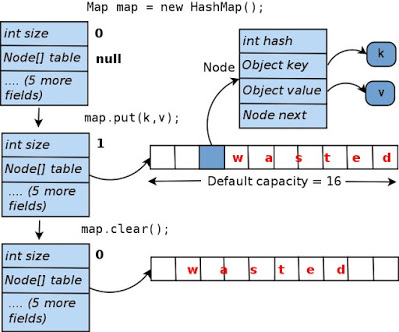
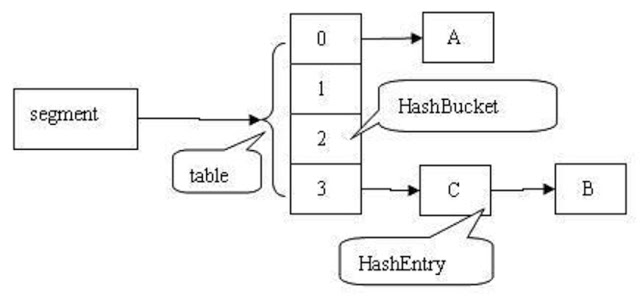
**ConcurrentHashMap**

Here are some of the best and frequently asked Java ConcurrentHashMap interview questions. These questions are collected from the real interview, hence, don't be surprised if you have already seen them during interviews.  
  
These questions will not only help you to do well on interviews but also encourage you to learn more about the concurrent hash map, which will eventually help you in your day-to-day programming job.  
  
  
**1. What is ConcurrentHashMap in Java?**   
The java.util.concurrent.ConcurrentHashMap is a concurrent collection class added on JDK 1.5 as a replacement of synchronized hash-based map implementations e.g. Hashtable and synchronized HashMap. They offer better performance and scalability over their synchronized counterpart with little risk.  
  
  
**2. Does ConcurrentHashMap thread-safe in Java?**   
Yes, ConcurrentHashMap is thread-safe in Java, which means two threads can modify the map without damaging its internal data structure e.g. [array](http://www.java67.com/2018/02/10-examples-of-array-in-java-tutorial.html) and [linked list](http://javarevisited.blogspot.sg/2017/07/top-10-linked-list-coding-questions-and.html#axzz4xXS86IVo). If you compare this to HashMap, which is not thread-safe, exposing HashMap to multiple threads **may damage internal data structure** and may render the map completely useless, where many links may go missing or pointing to wrong elements.  
  
  
**3. How does ConcurrentHashMap achieve thread-safety?**   
The java.util.ConcurrentHashMap achieves thread safety by dividing the map into segments and locking only the segment which requires instead of locking the whole map. So, yes, it achieves thread safety using locking but it performs better because, unlike HashMap, **it never locks the whole map**. This technique is also known as lock stripping.  
  
If you want to learn more about it, you can also take a look at the [**Java Collections from basics to Advanced courses**](https://click.linksynergy.com/deeplink?id=JVFxdTr9V80&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Fcollections-and-concurrent-collection-video-lectures-and-tutorials%2F) on Udemy.

[](https://click.linksynergy.com/deeplink?id=JVFxdTr9V80&mid=39197&murl=https://www.udemy.com/course/collections-and-concurrent-collection-video-lectures-and-tutorials/)

**4. Can multiple threads read from ConcurrentHashMap same time?**   
Yes, ConcurrentHashMap allows concurrent read without locking as reading operation doesn't require locking or thread-safety.  
  
  
**5. Can one thread read and the other writes on ConcurrentHashMap at the same time?**   
Yes, it's possible for a small number of the writer. For example, if a write operation is modifying one segment of ConcurrentHashmap and read operation is happening on other segments then a reader will not block, but if the reader thread is also trying to read from the same segment then it will block until the writer is done.  
  
  
**6. How does ConcurrentHashMap work internally?**   
The java.util.ConcurrentHashMap works similar to HashMap when it comes to storing key/value pairs and retrieving values. The only difference in its implementation comes from the concurrency perspective and how it achieves thread safety. It divides the map into several segments, by default 16, also known as synchronization level.  
  
Because of this, concurrent get(), put(), contains() operation is possible because it never locks the whole map but only the relevant segment is locked. This means readers can access the map concurrency with writers and a limited number of writers can modify the map concurrently. The result is better throughput and Scalability.  
  
You can further see [**The Complete Java MaseterClasss**](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https%3A%2F%2Fwww.udemy.com%2Fjava-the-complete-java-developer-course%2F) course on Udemy for details on the implementation of ConcurrentHashMap class. This course is recently updated for Java 11, the latest Java version as well.  
  
Here is a diagram that explains how a segment looks like in a ConcurrentHashMap of Java, basically it's nothing but a mini hash table with a bucket and a linked list of hash entries in case of collision:

[](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https://www.udemy.com/java-the-complete-java-developer-course/)

Since the Iterator returned by ConcurrentHashMap is weakly consistent, the recent concurrency modification may or may not be visible to it. There is no guarantee offered on such an operation.  
  
I also suggest you join the course [**Java Multithreading, Concurrency & Performance Optimization**](https://click.linksynergy.com/deeplink?id=JVFxdTr9V80&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Fjava-multithreading-concurrency-performance-optimization%2F) to learn more about how concurrency is handled by ConcurrentHashMap in Java.  
 **7. How do you atomically update a value in ConcurrentHashMap?**   
If you want to atomically update an existing value in ConcurrentHashMap, you can use the replace() function of concurrent hashmap.  
  
It accepts both old value and new value and only updates the map if the existing value in the map matches with the old value provided, this means the map is not concurrently modified during its call.  
  
If the existing value is changed and not match with the old value then replace fail with returning false. You can use call the replace() method in the while loop until you succeed, as shown below:

ConcurrentMap<String, Long> populationByCities = new ConcurrentHashMap<>();

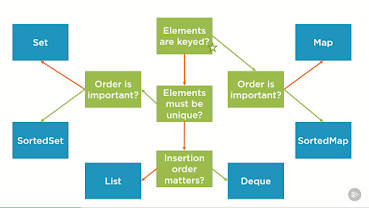
do{

Long currentValue = populationByCities.get("New York");

Long newValue = currentValue == null ? 1 : currentValue + 1;

}while(!populationByCities.replace("New York", currentValue, newValue));

You can also see the [**Java Collections: Fundamentals**](https://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fjava-collections-fundamentals) course by Richard Warburton, another Java Champion on Pluralsight to learn about basic Collection classes in Java.

[](https://javarevisited.blogspot.com/2020/04/top-5-courses-to-learn-java-collections-and-streams.html)

Btw, you would need a Pluralsight membership to access this course, which costs around $29 monthly or $299 annually. I have one and I also suggest all developers have that plan because Pluralsight is like NetFlix for Software developers. It has more than 5000+ good quality courses on all the latest topics. Since we programmers have to learn new things every day, an investment of $299 USD is not bad.  
  
Btw, it also offers a [**10-day free trial**](http://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Flearn) without any obligation which allows you to watch 200 hours of content. You can watch this course for free by signing for that trial.  
  
  
**8. How do you remove a mapping while iterating over ConcurrentHashMap?**   
You can use an Iterator to remove the mapping from ConcurrentHashMap in Java as shown below:

Map<String, Integer> bookAndPrice = new ConcurrentHashMap<>();

bookAndPrice.put("Effective Java", 42);

bookAndPrice.put("Head First Java", 29);

bookAndPrice.put("Java Concurrency in Practice", 33);

bookAndPrice.put("Head First Design Patterns", 41);

System.out.println("before removing : " + bookAndPrice);

Iterator<String> iterator = bookAndPrice.keySet().iterator();

while(iterator.hasNext()){

if(iterator.next().contains("Java")){

iterator.remove();

}

}

System.out.println("after removing : " + bookAndPrice);

Output

before removing : {Java Concurrency in Practice=33,

Head First Design Patterns=41, Effective Java=42, Head First Java=29}

after removing : {Head First Design Patterns=41}

**9. Does the Iterator of ConcurrentHashMap is fail-safe or fail-fast?**  
Iterator of ConcurrentHashMap is a fail-safe iterator which means it will not throw a ConcurrentModificationException, thus, eliminating the need to lock the map during iteration.  
  
The [Iterator](http://www.java67.com/2016/09/5-difference-between-iterator-and-ListIterator-in-java.html) returned by ConcurrentHashMap is also weakly consistent which means if the Map is modified during iteration, it may or may not reflect the recent modification. Generally, it creates a copy of the collection before iterating.  
  
**10. What will happen if you add a new mapping in ConcurrentHashMap while one thread is iterating over it?**   
This is one of the tricky questions related to ConcurrentHashMap. Since iterators of ConcurrentHashMap are weakly consistent and fail-safe they will not fail with [ConcurrentModificationException](http://javarevisited.blogspot.sg/2018/01/concurrentmodificationexception-while-removing-elements-from-ArrayList-in-loop-Java-solution.html#axzz57W3m5sdB) but it's also possible that they won't see any modification once iteration started. Even though it's implementation-dependent, JDK generally creates a separate copy of ConcurrentHashMap for iteration, instead of iterating over an original copy.  
  
**11. Can you pass an object of ConcurrentHahsMap when a Map is expected?**   
Yes because ConcurrentHashMap implements java.util.concurrent.ConcurrentMap interface, which extends java.util.Map interface, hence ConcurrentHashMap IS-A Map. Also, you can store an object of ConcurrentHashMap into a Map variable as shown below:

Map<String, Integer> bookAndPrice = new ConcurrentHashMap<>();

Though, this means, you may not have access to methods declared in the java.util.concurrent.ConcurrentHashMap class e.g. forEachKey() or forEachValue() method added in [Java 8](https://javarevisited.blogspot.com/2018/08/top-5-free-java-8-and-9-courses-for-programmers.html).  
  
  
That's all about some of the **frequently asked questions about ConcurrentHashMap on Java interviews**. These questions will not only help you to do well on your job interview but also encourage you to learn more about ConcurrentHashMap.  
  
Good and solid knowledge of ConcurrentHashMap is expected from both junior and senior Java developers given its importance and usability in every Java application.  
  
At least you should be comfortable with day-to-day operations with ConcurrentHashMap and understand how the internal implementation works, especially when compared to other thread-safe map implementations like Hashtable and Synchronized HashMap.

### **How Java HashMap works?**

HashMap is a key-value pair data structure. Each key will have a corresponding value and the key is the identifier for that value.

Internally, these key-value pairs are stored in logical blocks (buckets). When a pair is put in a HashMap, its key is used to compute hash code and that hash code identifies a bucket. Imagine it as an array index or a logical address or a door number. The key-value pair is stored at the bucket where the hash code points to. A bucket can have more than one key-value pairs stored in it.

When a value is looked upon using its key, first the hashcode is computed and the pointing bucket is reached. Then if that bucket has multiple pairs, then each of the ‘key’s are compared using the equals method to identify the matching pair.

Refer this tutorial to know about [what is hashcode and how it works](https://javapapers.com/core-java/hashcode-and-equals-methods-override/)? To know about buckets and hashing refer [Java Hashtable tutorial](https://javapapers.com/core-java/java-hashtable/).

### **What are fail-fast and fail-safe Iterators?**

fail-fast [Java iterators](https://javapapers.com/core-java/java-iterator/) may throw ConcurrentModifcationException if the underlying collection is modified during an iteration is in progress. fail-safe iterators will not throw any exception as the iteration happens on a clone of the instance. [fail fast and fail safe](https://javapapers.com/core-java/fail-fast-vs-fail-safe/) are paradigms that define how a system react when it encounters failure condition. Example for fail fast iterator is ArrayList and for fail safe iterator is ConcurrentHashMap.

### **What is BlockingQueue in Java?**

[Java BlockingQueue](https://javapapers.com/java/java-blockingqueue/) is a concurrent collection that is part of the util package. BlockingQueue is a type of queue which supports operations that wait for an element to become available when retrieving from it and similarly wait for a space to become available when storing elements in it. This collection is best used in a producer consumer scenario.

### **When do you use ConcurrentHashMap?**

In the above interview question number 2 we saw ConcurrentHashMap as an example for fail safe iterator. It allows complete concurrency for retrievals and updates. When there is a scenario where a high number of concurrent updates are expected then ConcurrentHashMap can be used. This is very similar to a Hashtable but does not lock the entire table to provide concurrency and so it is better performance point of view. When there are high number of updates and less number of read concurrently, then ConcurrentHashMap should be used.

### **Which List implementation provides fastest insertion?**

This is between LinkedList and ArrayList. These two are different variants of List implementations. LinkedList is a doubly linked list datastructure and ArrayList is dynamically resizing array. Performance of these two collections with respect to insert is O(n) for LinkedList and O(n-index) for ArrayList.

In LinkedList the cost is always a constant factor, it is about allocating a node and linking with adjacent elements. In ArrayList the cost varies based on whether the insertion is at beginning or at the end of the list. Other elements already existing in the ArrayList should be adjusted for positions according to the insertion.

If the insertion is at the end of the List then ArrayList is faster than LinkedList. If the insertion is at the beginning and also if the list is longer then LinkedList wins.

### **Difference between Iterator and ListIterator**

ListIterator is used to traverse List type of collections exclusively where in Iterator can be used traverse any type of collections. ListIterator has got additional features over an Iterator and they are,

* + ListIterator can traverse a List backwards where in Iterator cannot do.
  + Using ListIterator an element can be added at any given point.
  + Get the current index at any traversal moment.
  + Replace an element at the traversal point.

### **What is CopyOnWriteArrayList, how it is different than ArrayList?**

CopyOnWriteArrayList is a thread-safe counterpart of ArrayList. All mutable operations like add and set are implement by using a copy of the underlying array. Write operation is slower when compared the ArrayList as it takes a snapshot of the instance and does the write. This is useful when the traversal of the collection need not be synchronized during traversal with the original instance and the traversal is larger in count than the updates. This provides a fail safe iterator as it does not throw exception when the underlying collection is modified. At the same time the iterator will not reflect the modifications done to the collection, it just shows the snapshot of state taken at the moment when the iterator is created.

### **Difference between Iterator and Enumeration**

If the interviewer asks this question in a Java interview any more, consider him to be legacy. Mainly Iterator is different from Enumeration in two ways,

* + Iterator allows the removal of elements from the underlying collection.
  + Method names are standardized in Iterator.

Iterator is brought in as a replacement for Enumeration in [Java 1.2 release](https://javapapers.com/core-java/java-history/). Use Iterator everywhere instead of Enumeration

### **How a HashMap can be synchronized?**

There are two options when we need a synchronized HashMap.

* + Use Collections.synchronizedMap(..) to synchronize the HashMap.
  + Use ConcurrentHashMap.

The preferred choice between these two options is to use the ConcurrentHashMap. That’s because we need not lock the whole object and ConcurrentHashMap partitions the map and obtains lock as necessary. Read the interview question number 4 above. Need not reinvent the wheel unless you are going to provide a different and greater implementation than the ConcurrentHashMap.

### **Difference between IdentityHashMap and HashMap**

IdentityHashMap is an implementation of Map interface. Unlike HashMap, this uses reference equality. That means,

* + in HashMap, two elements are equal if key1.equals(key2)
  + iin IdentityHashMap, two elements are equal if key1 == key2

Map imposes a contract to honor, the implementations should use object equality. If equals of method returns same value for two keys, then the hash code value should be same.

* + IdentityHashMap intentionally violates the contract and does reference equality.
  + For hashing, the IdentityHashMap uses System.identityHashCode(object) instead of hashCode() as done by HashMap.
  + IdentityHashMap is relatively faster than HashMap for operations.
  + Keys are mutable in IdentityHashMap but in HashMap keys are mutable.

This is second part of a series of Java collections interview questions and the third part will be posted soon.

# **What Is the Purpose of the Initial Capacity and Load Factor Parameters of a Hashmap? What Are Their Default Values?**

The initialCapacity argument of the HashMap constructor affects the size of the internal data structure of the HashMap, but reasoning about the actual size of a map is a bit tricky. The HashMap‘s internal data structure is an array with the power-of-two size. So the initialCapacity argument value is increased to the next power-of-two (for instance, if you set it to 10, the actual size of the internal array will be 16).

The load factor of a HashMap is the ratio of the element count divided by the bucket count (i.e. internal array size). For instance, if a 16-bucket HashMap contains 12 elements, its load factor is 12/16 = 0.75. A high load factor means a lot of collisions, which in turn means that the map should be resized to the next power of two. So the loadFactor argument is a maximum value of the load factor of a map. When the map achieves this load factor, it resizes its internal array to the next power-of-two value.

The initialCapacity is 16 by default, and the loadFactor is 0.75 by default, so you could put 12 elements in a HashMap that was instantiated with the default constructor, and it would not resize. The same goes for the HashSet, which is backed by a HashMap instance internally.

# **How Is Hashmap Implemented in Java? How Does Its Implementation Use Hashcode and Equals Methods of Objects? What Is the Time Complexity of Putting and Getting an Element from Such Structure?**

The HashMap class represents a typical hash map data structure with certain design choices.

The HashMap is backed by a resizable array that has a size of power-of-two. When the element is added to a HashMap, first its hashCode is calculated (an int value). Then a certain number of lower bits of this value are used as an array index. This index directly points to the cell of the array (called a bucket) where this key-value pair should be placed. Accessing an element by its index in an array is a very fast O(1) operation, which is the main feature of a hash map structure.

A hashCode is not unique, however, and even for different hashCodes, we may receive the same array position. This is called a collision. There is more than one way of resolving collisions in the hash map data structures. In Java's HashMap, each bucket actually refers not to a single object, but to a red-black tree of all objects that landed in this bucket (prior to Java 8, this was a linked list).

So when the HashMap has determined the bucket for a key, it has to traverse this tree to put the key-value pair in its place. If a pair with such key already exists in the bucket, it is replaced with a new one.

To retrieve the object by its key, the HashMap again has to calculate the hashCode for the key, find the corresponding bucket, traverse the tree, call equals on keys in the tree and find the matching one.

HashMap has O(1) complexity, or constant-time complexity, of putting and getting the elements. Of course, lots of collisions could degrade the performance to O(log(n)) time complexity in the worst case, when all elements land in a single bucket. This is usually solved by providing a good hash function with a uniform distribution.

When the HashMap internal array is filled (more on that in the next question), it is automatically resized to be twice as large. This operation infers rehashing (rebuilding of internal data structures), which is costly, so you should plan the size of your HashMap beforehand.

# **What Is the Difference Between Hashset and Treeset?**

Both **HashSet** and **TreeSet** classes implement the Set interface and represent sets of distinct elements. Additionally, TreeSet implements the NavigableSet interface. This interface defines methods that take advantage of the ordering of elements.

HashSet is internally based on a HashMap, and TreeSet is backed by a TreeMap instance, which defines their properties: HashSet does not keep elements in any particular order. Iteration over the elements in a HashSet produces them in a shuffled order. TreeSet, on the other hand, produces elements in order according to some predefined Comparator.

# **Explain the Difference Between Linkedlist and Arraylist.**

**ArrayList** is an implementation of the List interface that is based on an array. ArrayList internally handles resizing of this array when the elements are added or removed. You can access its elements in constant time by their index in the array. However, inserting or removing an element infers shifting all consequent elements which may be slow if the array is huge and the inserted or removed element is close to the beginning of the list.

**LinkedList** is a doubly-linked list: single elements are put into Node objects that have references to previous and next Node. This implementation may appear more efficient than ArrayList if you have lots of insertions or deletions in different parts of the list, especially if the list is large.

In most cases, however, ArrayList outperforms LinkedList. Even elements shifting in ArrayList, while being an O(n) operation, is implemented as a very fast System.arraycopy() call. It can even appear faster than the LinkedList‘s O(1) insertion which requires instantiating a Node object and updating multiple references under the hood. LinkedList also can have a large memory overhead due to a creation of multiple small Node objects.