java.util

## Class HashMap<K,V>

* [java.lang.Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)
  + [java.util.AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<K,V>
    - java.util.HashMap<K,V>
* **Type Parameters:**

K - the type of keys maintained by this map

V - the type of mapped values

**All Implemented Interfaces:**

[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html), [Cloneable](https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html" \o "interface in java.lang), [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<K,V>

**Direct Known Subclasses:**

[LinkedHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html), [PrinterStateReasons](https://docs.oracle.com/javase/8/docs/api/javax/print/attribute/standard/PrinterStateReasons.html" \o "class in javax.print.attribute.standard)

### *Nested Class Summary*

### Nested classes/interfaces inherited from class java.util.[AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)

[AbstractMap.SimpleEntry](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.SimpleEntry.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.SimpleEntry.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.SimpleEntry.html)>, [AbstractMap.SimpleImmutableEntry](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.SimpleImmutableEntry.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.SimpleImmutableEntry.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.SimpleImmutableEntry.html)>

### Nested classes/interfaces inherited from interface java.util.[Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)

[Map.Entry](https://docs.oracle.com/javase/8/docs/api/java/util/Map.Entry.html)<[K](https://docs.oracle.com/javase/8/docs/api/java/util/Map.Entry.html),[V](https://docs.oracle.com/javase/8/docs/api/java/util/Map.Entry.html)>

|  |
| --- |
| **Constructors** |
| **Constructor and Description** |
| [**HashMap**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#HashMap--)()  Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75). |
| [**HashMap**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#HashMap-int-)(int initialCapacity)  Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75). |
| [**HashMap**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#HashMap-int-float-)(int initialCapacity, float loadFactor)  Constructs an empty HashMap with the specified initial capacity and load factor. |
| [**HashMap**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#HashMap-java.util.Map-)([**Map**](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<? extends [**K**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html),? extends [**V**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)> m)  Constructs a new HashMap with the same mappings as the specified Map. |

public class **HashMap<K,V>**

extends [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<K,V>

implements [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<K,V>, [Cloneable](https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html), [Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html)

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

An instance of HashMap has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is *rehashed* (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

If many mappings are to be stored in a HashMap instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table. Note that using many keys with the same hashCode() is a sure way to slow down performance of any hash table. To ameliorate impact, when keys are [Comparable](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html), this class may use comparison order among keys to help break ties.

How HashMap works in java

Most common interview questions are “How HashMap works in java”, “How get and put method of HashMap work internally”. Here I am trying to explain internal functionality with an easy example. Rather than going through theory, we will start with example first, so that you will get better understanding and then we will see how get and put function work in java.  
Lets take a very simple example. I have a Country class, we are going to use Country class object as key and its capital name(string) as value. Below example will help you to understand, how these key value pair will be stored in hashmap.

## 1. Country.java

|  |  |
| --- | --- |
|  | package org.arpit.javapostsforlearning; |
|  | public class Country { |

|  |  |
| --- | --- |
|  |  |
|  | String name; |

|  |  |
| --- | --- |
| 05 | long population; |
| 06 |  |

|  |  |
| --- | --- |
|  | public Country(String name, long population) { |
|  | super(); |

|  |  |
| --- | --- |
|  | this.name = name; |
|  | this.population = population; |

|  |  |
| --- | --- |
| 11 | } |
| 12 | public String getName() { |

|  |  |
| --- | --- |
| 13 | return name; |
| 14 | } |

|  |  |
| --- | --- |
| 15 | public void setName(String name) { |
| 16 | this.name = name; |

|  |  |
| --- | --- |
| 17 | } |
| 18 | public long getPopulation() { |

|  |  |
| --- | --- |
| 19 | return population; |
| 20 | } |

|  |  |
| --- | --- |
| 21 | public void setPopulation(long population) { |
| 22 | this.population = population; |

|  |  |
| --- | --- |
| 23 | } |
| 24 |  |

|  |  |
| --- | --- |
| 25 | // If length of name in country object is even then return 31(any random number) and if odd then return 95(any random number). |
| 26 | // This is not a good practice to generate hashcode as below method but I am doing so to give better and easy understanding of hashmap. |

|  |  |
| --- | --- |
| 27 | @Override |
| 28 | public int hashCode() { |

|  |  |
| --- | --- |
| 29 | if(this.name.length()%2==0) |
| 30 | return 31; |

|  |  |
| --- | --- |
| 31 | Else |
|  | return 95; |

|  |  |
| --- | --- |
| 33 | } |
| 34 | @Override |

|  |  |
| --- | --- |
| 35 | public boolean equals(Object obj) { |
| 36 |  |

|  |  |
| --- | --- |
| 37 | Country other = (Country) obj; |
| 38 | if (name.equalsIgnoreCase((other.name))) |

|  |  |
| --- | --- |
| 39 | return true; |
| 40 | return false; |

|  |  |
| --- | --- |
| 41 | } |
| 42 |  |

|  |  |
| --- | --- |
| 43 | } |

If you want to understand more about hashcode and equals method of object, you may refer **[hashcode() and equals() method in java](http://javapostsforlearning.blogspot.in/2014/02/hashcode-and-equals-method-in-java.html" \t "_blank)**

## 2. HashMapStructure.java(main class)

|  |  |
| --- | --- |
| 01 | import java.util.HashMap; |
| 02 | import java.util.Iterator; |

|  |  |
| --- | --- |
| 03 |  |
| 04 | public class HashMapStructure { |

|  |  |
| --- | --- |
| 05 |  |
| 06 | /\*\* |

|  |  |
| --- | --- |
| 07 | \* @author Arpit Mandliya |
| 08 | \*/ |

|  |  |
| --- | --- |
| 09 | public static void main(String[] args) { |
| 10 |  |

|  |  |
| --- | --- |
| 11 | Country india=new Country("India",1000); |
| 12 | Country japan=new Country("Japan",10000); |

|  |  |
| --- | --- |
| 13 |  |
| 14 | Country france=new Country("France",2000); |

|  |  |
| --- | --- |
| 15 | Country russia=new Country("Russia",20000); |
| 16 |  |

|  |  |
| --- | --- |
| 17 | HashMap<country,string> countryCapitalMap=new HashMap<country,string>(); |
| 18 | countryCapitalMap.put(india,"Delhi"); |

|  |  |
| --- | --- |
| 19 | countryCapitalMap.put(japan,"Tokyo"); |
| 20 | countryCapitalMap.put(france,"Paris"); |

|  |  |
| --- | --- |
| 21 | countryCapitalMap.put(russia,"Moscow"); |
| 22 |  |

|  |  |
| --- | --- |
| 23 | Iterator<country> countryCapitalIter=countryCapitalMap.keySet().iterator();//put debug point at this line |
| 24 | while(countryCapitalIter.hasNext()) |

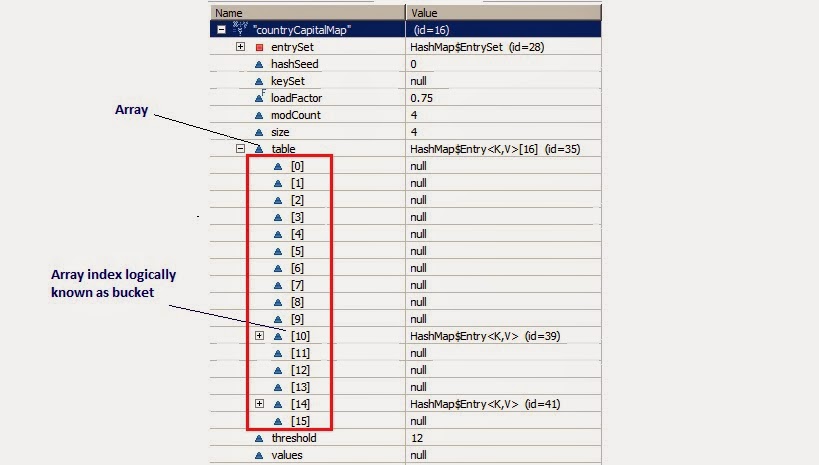
|  |  |
| --- | --- |
| 25 | { |
| 26 | Country countryObj=countryCapitalIter.next(); |

|  |  |
| --- | --- |
| 27 | String capital=countryCapitalMap.get(countryObj); |
| 28 | System.out.println(countryObj.getName()+"----"+capital); |

|  |  |
| --- | --- |
| 29 | } |
| 30 | } |

|  |  |
| --- | --- |
| 31 |  |
| 32 |  |

|  |  |
| --- | --- |
| 33 | } |
| 34 | </country></country,string></country,string> |

Now put debug point at line 23 and right click on project->debug as-> java application. Program will stop execution at line 23 then right click on countryCapitalMap then select watch.You will be able to see structure as below.  
   
[](http://www.javacodegeeks.com/wp-content/uploads/2014/03/HashMapStructure1.jpg)  
Now From above diagram, you can observe following points

1. There is an Entry[] array called table which has size 16.
2. This table stores Entry class’s object. HashMap class has a inner class called Entry.This Entry have key value as instance variable. Lets see structure of entry class Entry Structure.

|  |  |
| --- | --- |
| 1 | static class Entry implements Map.Entry |
| 2 | { |

|  |  |
| --- | --- |
| 3 | final K key; |
| 4 | V value; |

|  |  |
| --- | --- |
| 5 | Entry next; |
| 6 | final int hash; |

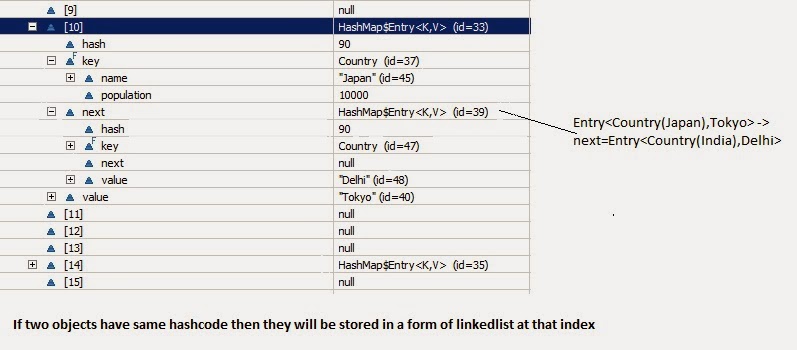
|  |  |
| --- | --- |
| 7 | ...//More code goes here |
| 8 | } |

1. Whenever we try to put any key value pair in hashmap, Entry class object is instantiated for key value and that object will be stored in above mentioned Entry[](table). Now you must be wondering, where will above created Enrty object get stored(exact position in table). The answer  is, hash code is calculated for a key by calling Hascode() method. This hashcode is used to calculate index for above Entry[] table.
2. Now, If you see at array index 10 in above diagram, It has an Entry object named HashMap$Entry.
3. We have put 4 key-values in hashmap but it seems to have only 2!!!!This is because if two objects have same hashcode, they will be stored at same index. Now question arises how? It stores objects in a form of LinkedList(logically).

So how hashcode of above country key-value pairs are calculated.

|  |  |
| --- | --- |
| 1 | Hashcode for Japan = 95 as its length is odd. |
| 2 | Hashcode for India =95 as its length is odd |

|  |  |
| --- | --- |
| 3 | HashCode for Russia=31 as its length is even. |
| 4 | HashCode for France=31 as its length is even. |

Below diagram will explain LinkedList concept clearly.  
   
[](http://www.javacodegeeks.com/wp-content/uploads/2014/03/HashMapStructure2.jpg)  
So now if you have good understanding of hashmap structure,Lets go through put and get method.

## Put :

Lets see implementation of put method:

|  |  |
| --- | --- |
| 01 | /\*\* |
| 02 | \* Associates the specified value with the specified key in this map. If the |

|  |  |
| --- | --- |
| 03 | \* map previously contained a mapping for the key, the old value is |
| 04 | \* replaced. |

|  |  |
| --- | --- |
| 05 | \* |
| 06 | \* @param key |

|  |  |
| --- | --- |
| 07 | \*            key with which the specified value is to be associated |
| 08 | \* @param value |

|  |  |
| --- | --- |
| 09 | \*            value to be associated with the specified key |
| 10 | \* @return the previous value associated with <tt>key</tt>, or <tt>null</tt> |

|  |  |
| --- | --- |
| 11 | \*         if there was no mapping for <tt>key</tt>. (A <tt>null</tt> return |
| 12 | \*         can also indicate that the map previously associated |

|  |  |
| --- | --- |
| 13 | \*         <tt>null</tt> with <tt>key</tt>.) |
| 14 | \*/ |

|  |  |
| --- | --- |
| 15 | public V put(K key, V value) { |
| 16 | if (key == null) |

|  |  |
| --- | --- |
| 17 | return putForNullKey(value); |
| 18 | int hash = hash(key.hashCode()); |

|  |  |
| --- | --- |
| 19 | int i = indexFor(hash, table.length); |
| 20 | for (Entry<k , V> e = table[i]; e != null; e = e.next) { |

|  |  |
| --- | --- |
| 21 | Object k; |
| 22 | if (e.hash == hash && ((k = e.key) == key || key.equals(k))) { |

|  |  |
| --- | --- |
| 23 | V oldValue = e.value; |
| 24 | e.value = value; |

|  |  |
| --- | --- |
| 25 | e.recordAccess(this); |
| 26 | return oldValue; |

|  |  |
| --- | --- |
| 27 | } |
| 28 | } |

|  |  |
| --- | --- |
| 29 |  |
| 30 | modCount++; |

|  |  |
| --- | --- |
| 31 | addEntry(hash, key, value, i); |
| 32 | return null; |

|  |  |
| --- | --- |
| 33 | } |

now lets understand above code step by step

1. Key object is checked for null. If key is null then it will be stored at table[0] because hashcode for null is always 0.
2. Key object’s hashcode() method is called and hash code is calculated. This hashcode is used to find index of array for storing Entry object. It may happen sometimes that, this hashcode function is poorly written so JDK designer has put another function called hash() which takes above calculated hash value as argument.If you want to learn more about hash() function, you can refer [hash and indexFor method in hashmap](http://javapostsforlearning.blogspot.in/2014/02/hash-and-indexfor-method-in-hashmap.html).
3. indexFor(hash,table.length)  is used to calculate exact index in table array for storing the Entry object.
4. As we have seen in our example, if two key objects have same hashcode(which is known as **collision**) then it will be stored in form of linkedlist. So here, we will iterate through our linkedlist.

* If there is no element present at that index which we have just calculated then it will directly put our Entry object at that index.
* If There is element present at that index then it will iterate until it gets Entry->next as null.Then current Entry object become next node in that linkedlist
* What if we are putting same key again, logically it should replace old value. Yes,it will do that.While iterating it will check key equality by calling equals() method(**key.equals(k)**), if this method returns true then it replaces value object with current Entry’s value object.

## Get:

Lets see implementation of get now:

|  |  |
| --- | --- |
| 01 | /\*\* |
| 02 | \* Returns the value to which the specified key is mapped, or {@code null} |

|  |  |
| --- | --- |
| 03 | \* if this map contains no mapping for the key. |
| 04 | \* |

|  |  |
| --- | --- |
| 05 | \* <p> |
| 06 | \* More formally, if this map contains a mapping from a key {@code k} to a |

|  |  |
| --- | --- |
| 07 | \* value {@code v} such that {@code (key==null ? k==null : |
| 08 | \* key.equals(k))}, then this method returns {@code v}; otherwise it returns |

|  |  |
| --- | --- |
| 09 | \* {@code null}. (There can be at most one such mapping.) |
| 10 | \* |

|  |  |
| --- | --- |
| 11 | \* </p><p> |
| 12 | \* A return value of {@code null} does not <i>necessarily</i> indicate that |

|  |  |
| --- | --- |
| 13 | \* the map contains no mapping for the key; it's also possible that the map |
| 14 | \* explicitly maps the key to {@code null}. The {@link #containsKey |

|  |  |
| --- | --- |
| 15 | \* containsKey} operation may be used to distinguish these two cases. |
| 16 | \* |

|  |  |
| --- | --- |
| 17 | \* @see #put(Object, Object) |
| 18 | \*/ |

|  |  |
| --- | --- |
| 19 | public V get(Object key) { |
| 20 | if (key == null) |

|  |  |
| --- | --- |
| 21 | return getForNullKey(); |
| 22 | int hash = hash(key.hashCode()); |

|  |  |
| --- | --- |
| 23 | for (Entry<k , V> e = table[indexFor(hash, table.length)]; e != null; e = e.next) { |
| 24 | Object k; |

|  |  |
| --- | --- |
| 25 | if (e.hash == hash && ((k = e.key) == key || key.equals(k))) |
| 26 | return e.value; |

|  |  |
| --- | --- |
| 27 | } |
| 28 | return null; |

|  |  |
| --- | --- |
| 29 | } |

As you got the understanding on put functionality of hashmap. So to understand get functionality is quite simple. If you pass any key to get value object from hashmap.

1. Key object is checked for null. If key is null then value of Object resides at table[0] will be returned.
2. Key object’s hashcode() method is called and hash code is calculated.
3. indexFor(hash,table.length)  is used to calculate exact index in table array using generated hashcode for getting the Entry object.
4. After getting index in table array, it will iterate through linkedlist and check for key equality by calling equals() method and if it returns true then it returns the value of Entry object else returns null.

## Key points to Remeber:

* HashMap has a inner class called Entry which stores key-value pairs.
* Above Entry object is stored in Entry[ ](Array) called table
* An index of table is logically known as bucket and it stores first element of linkedlist
* Key object’s hashcode() is used to find bucket of that Entry object.
* If two key object ‘s have same hashcode , they will go in same bucket of table array.
* Key object ‘s equals() method is used to ensure uniqueness of key object.
* Value object  ‘s equals() and hashcode() method is not used at all

**How is HashMap internally implemented?**  
pair is stored in HashMap. Pair is an entry to hashMap. Entries are stored in an array. So we have an array of entries.   
Each Entry has a key. We calculate hash(key.hashcode()), which determines the index in array. hash method shortens the hashcode value to a valid int index.   
The value at any index in array is called bucket, which holds the Entry.   
If for two Entries index comes out to be same, while storing. This is possible if two keys have same hashCode value. Then those two entries are stored in same bucket. Bucket is a linkedlist ate any index in array, and bucket holds the Entries. So we can have multiple entries in same bucket.   
When get(Key) operation is called. We do following -   
1. Calculate the hash(key.hashcode), so that we know the index of the bucket in array.   
2. If bucket holds more than one Key, use equals(on key) to return the exact match.

======================Another Article==============

How a HashMap works internally has become a popular question in almost all the interview. As almost everybody knows how to use a HashMap or the [**difference between HashMap and Hashtable**](http://www.javainterviewpoint.com/differences-betwen-hashmap-hashtable/). However, many people fail when the question is "How does a HashMap work internally?"

So the answer to the question is that it works based **on the hashing principle,** but it is not as simple as it sounds. Hashing is the mechanism of assigning unique code to a variable or attribute using an algorithm to enable easy retrieval. A true hashing mechanism should always return the same hashCode() when it is applied to the same object.

Then comes the question, "How does hashing help in storing and retrieving the value in HashMap?" Many will say that the value will be stored in the bucket and retrieved using the key. If you think that is how it works then you are absolutely wrong. To prove it, let's take a look at the HashMap class:

/\*\*

\* The table, resized as necessary. Length MUST Always be a power of two.

\*/

transient Entry[] table;

So what is the use of Entry[] in a HashMap for? Because the HashMap stores the Objects as **Entry instances,**not as**key and value**

## What Is Entry Class?

HashMap has an inner class called an Entry Class which holds the key and values. And there is also something called next, which you will get to know a bit later.

static class Entry<K,V> implements Map.Entry<K,V>

{

final K key;

V value;

Entry<K,V> next;

final int hash;

........

}

You know that the HashMap stores the Entry instances in an array and not as key-value pairs. In order to store a value, you will use the put() method of the HashMap, so now let's dig into that and see how it works.

## How Does Put() Method Work Internally?

The Code [**implementation of the put() method**](https://dzone.com/articles/www.javainterviewpoint.com/) will look like this:

public V put(K key, V value)

{

if (key == null)

return putForNullKey(value);

int hash = hash(key.hashCode());

int i = indexFor(hash, table.length);

for (Entry<K,V> e = table[i]; e != null; e = e.next)

{

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

{

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

return null;

}

* First, it checks if the key given is null or not. If the given key is null, it will be stored in the zero position, as the hashcode of null will be zero.
* Then it applies the hashcode to the key **.hashCode()** by calling the hashcode method. In order to get the value within the limits of an array, the hash(key.hashCode()) is called, which performs some shifting operations on the hashcode.
* The **indexFor()** method is used to get the exact location to store the Entry object.
* Then comes the most important part what happens if two different object has the same hashcode( eg : Aa,BB will have the same hashcode) and will it be stored in the same bucket. To handle this let's think of the LinkedList in data structure it will have a next attribute which will always point to the next object. The same way the next attribute in the Entry class points to the next object. Using this different objects with the same hashcode will be placed next to each other.
* In the case of the Collision, the HashMap checks for the value of the next attribute if it is **null** it inserts the Entry object in that location, if next attribute is not null then it keeps the loop running till next attribute is null then stores the Entry object there.

## How are Duplicate Keys Prevented in HashMap?

As we all know, HashMap doesn't allow duplicate keys, even though when we insert the same key with different values, only the latest value is returned.

import java.util.HashMap;

import java.util.Map;

public class HashMapEg

{

public static void main(String[] args)

{

Map map = new HashMap();

map.put(1,"sam");

map.put(1,"Ian");

map.put(1,"Scott");

map.put(null,"asdf");

System.out.println(map);

}

}

For the above code, you will get the output as  {null=asdf, 1=Scott} ,  as the values sam  and Ian  will be replaced by Scott. So, how does this happen?

All the Entry Objects in the LinkedList will have the same hashcode, but HashMap uses equals() . This method checks the equality, so if **key.equals(k)**is true, then it will replace the value object inside the Entry class and not the key. So this way it prevents the duplicate key from being inserted.

## How Does Get() Method Work Internally?

Almost the same logic as applied in the put() method will be used to retrieve the value.

public V get(Object key)

{

if (key == null)

return getForNullKey();

int hash = hash(key.hashCode());

for (Entry<K,V> e = table[indexFor(hash, table.length)];e != null;e = e.next)

{

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

return e.value;

}

return null;

}

* First, it gets the hash code of the key object, which is passed, and finds the bucket location.
* If the correct bucket is found, it returns the value (e.value)
* If no match is found, it returns null.

## What Happens If Two Keys Have the Same Hashcode?

The same collision resolution mechanism will be used here. **key.equals(k)**will check until it is true, and if it is true, it returns the value of it.

Hope this article clarifies the troublesome HashMap internal mechanism. Happy Learning !! :)

How ConcurrentHashMap Internally Works In Java With Example

ConcurrentHashMap utilizes the same principles of HashMap, but is designed primarily for a multi-threaded application and hence it does not require explicit synchronization.   The only thread safe collection objects were Hashtable and synchronized Map prior to JDK 5

**ConcurrentHashMap:** It allows concurrent access to the map. Part of the map called Segment (internal data structure) is only getting locked while adding or updating the map. So ConcurrentHashMap allows concurrent threads to read the value without locking at all. This data structure was introduced to improve performance..

**Interviewer : Why we need ConcurrentHashMap when we already had Hashtable ?**  
  
Hashtable provides concurrent access to the Map. Entries objects by locking the entire map to perform any sort of operation (update,delete,read,create). Suppose we have a web application, the overhead created by Hashtable  (locking the entire map) can be ignored under normal load. But under heavy load , the overhead of locking the entire map may prove fatal and may lead to delay response time and  overtaxing of the server.

This  is where ConcurrentHashMap comes to rescue. According to [ConcurrentHashMap Oracle docs,](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html" \t "_blank)  
ConcurrentHashMap class is fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization details. So the main purpose of this class is to provide the same functionality as of Hashtable but with a performance comparable to HashMap.   
  
ConcurrentHashMap achieves this by a simple tweak. So this leads to our main question

**How ConcurrentHashMap works in Java**  
  
According to ConcurrentHashMap Oracle docs,   
  
The constructor of ConcurrentHashMap looks like this :  
  
**public ConcurrentHashMap*(int initialCapacity, float loadFactor, int concurrencyLevel)***  
  
So the above line  creates a new, empty map with the specified initial capacity, load factor and concurrency level.

where,

**Important Parameters to consider from ConcurrentHashMap Constructor:**

**initialCapacity** - the initial capacity. The implementation performs *internal sizing to accommodate this many elements.*

A ConcurrentHashMap is divided into number of segments, and the example which I am explaining here used default as 32 on initialization.

A ConcurrentHashMap has internal final class called Segment so we can say that ConcurrentHashMap is internally divided in segments of size 32, so at max 32 threads can work at a time. It means each thread can work on a each segment during high concurrency and atmost 32 threads can operate at max which simply maintains 32 locks to guard each bucket of the ConcurrentHashMap.

The definition of Segment is as below:

/\*\* Inner Segment class plays a significant role \*\*/

protected static final class Segment {

protected int count;

protected synchronized int getCount() {

return this.count;

}

protected synchronized void synch() {}

}

/\*\* Segment Array declaration \*\*/

public final Segment[] segments = new Segment[32];

**concurrencyLevel** - the estimated number of concurrently updating threads. The implementation performs *internal sizing to try to accommodate this many threads.*

**====================================================**

**How Segment array size is tuned?**

Segment size decides the number of Threads that can parallelly write to a map.  
Segment array size is configured using ConcurrencyLevel parameter as shown below,

ConcurrentHashMap m = **new**  ConcurrentHashMap(initialCapacity, loadFactor, concurrencyLevel)

**Example:**

ConcurrentHashMap m = **new** ConcurrentHashMap(200 , 0.75f, 10);

**Initial capacity** is 200, it means ConcurrentHashMap make sure it has space for adding 200 key-value pairs after creation.

**Load factor**is 0.75, it means when average number of elements per map exceeds 150 (intital capacity \* load factor = 200 \* 0.75 = 150) at that time map size will be increased and existing items in map are rehashed to put in new larger size map.  
For more details on Load Factor: [**Load factor in Map**](http://javabypatel.blogspot.in/2015/10/what-is-load-factor-and-rehashing-in-hashmap.html)

**Concurrency level**is 10, it means at any given point of time Segment array size will be 10 or greater than 10, so that 10 threads can able to parallelly write to a map.

==========================================================  
In the [ConcurrentHashMap Api](http://www.docjar.com/html/api/java/util/concurrent/ConcurrentHashMap.java.html" \t "_blank) , you will find the following constants.

static final int DEFAULT\_INITIAL\_CAPACITY = 16;

static final int DEFAULT\_CONCURRENCY\_LEVEL = 16;

initial capacity parameter and concurrency level parameters of ConcurrentHashMap constructor (or Object) are  set to 16 by default.  
  
Thus, instead of a map wide lock, ConcurrentHashMap maintains  a list of 16 locks by default ( number of locks equal to the initial capacity , which is by default  16) each of which is used to lock on a single bucket of the Map.This indicates that 16 threads (number of threads equal to the concurrency level , which is by  default 16) can modify the collection at the same time , given ,each thread works on different bucket. So unlike hashtable, we perform any sort of operation ( update ,delete ,read ,create) without locking on entire map in ConcurrentHashMap.  
  
Retrieval operations (including get) generally do not block so may overlap with update operations (including put and remove). Retrievals reflect the results of the most recently *completed* update operations holding upon their onset.   
  
The allowed concurrency among update operations is guided by the optional concurrencyLevel constructor argument (default 16), which is used as a hint for internal sizing. The table is internally partitioned to try to permit the indicated number of concurrent updates without contention. Because placement in hash tables is essentially random, the actual concurrency will vary. Ideally, you should choose a value to accommodate as many threads as will ever concurrently modify the table. Using a significantly higher value than you need can waste space and time, and a significantly lower value can lead to thread contention  
  
**Interviewer : Can two threads update the ConcurrentHashMap simultaneously ?**  
  
                                                       
Yes it is possible that two threads can simultaneously write on the ConcurrentHashMap. ConcurrentHashMap default implementation allows 16 threads to read and write in parallel.   
But in the worst case scenario , when two objects lie in the same segment or same partition, then parallel write would not be possible.

**Interviewer : Why ConcurrentHashMap does not allow null keys and null values ?**  
  
According to the [author of the ConcurrentHashMap (Doug lea himself)](http://cs.oswego.edu/pipermail/concurrency-interest/2006-May/002485.html)

The main reason that nulls aren't allowed in ConcurrentMaps (ConcurrentHashMaps, ConcurrentSkipListMaps) is that ambiguities that may be just barely tolerable in non-concurrent maps can't be accommodated. The main one is that if map.get(key) returns null, you can't detect whether the key explicitly maps to null vs the key isn't mapped. In a non-concurrent map, you can check this via map.contains(key), but in a concurrent one, the map might have changed between   
calls.  
  
In simple words,   
  
The code is like this :   
 

**if** (map.containsKey(k)) {

**return** map.get(k);

} **else** {

**throw** **new** **KeyNotPresentException**();

}

It might be possible that key k might be deleted in between the get(k) and containsKey(k) calls. As a result , the code will return null as opposed to KeyNotPresentException (Expected Result if key is not present).