Hashing :How Hash Map Works In Java Or How Get() Method Works Internally

One of the most darling question of the core java interviewers is How hash map works in java or internal.implementation of hashmap. Most of the candidates rejection chances increases if the candidate do not give the satisfactory explanation . This question shows that candidate has good knowledge of Collection . So this question should be in your to do list before appearing for the interview.

Read also How Hashset works in java or How it ensures uniqueness in java

How Hashmap works in Java

HashMap works on the principle of Hashing . To understand Hashing , we should understand the three terms first i.e *Hash Function , Hash Value and Bucket .*

What is Hash Function , Hash Value and Bucket ?

hashCode() function which returns an integer value is the **Hash function**. The important point to note that , this method is present in Object class (Mother of all class) .

This is the code for the hash function(also known as hashCode method) in Object Class :

```
public native int hashCode();
```

The most important point to note from the above line : hashCode method return int value .
So the **Hash value** is the int value returned by the hash function **.**

So summarize the terms in the diagram below :

What is bucket ?
A bucket is used to store key value pairs . A bucket can have multiple key-value pairs . In hash map, bucket used simple linked list to store objects .

After understanding the terms we are ready to move next step , **How hash map works in java or How get() works internally in java .**

Code inside Java Api (HashMap class internal implementation) for HashMap get(Obejct key) method

```
Public  V get(Object key)
 {
    if (key ==null)
3.    //Some code

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

5.    // if key found in hash table then  return value
6.    //    else return null
  }
```

Hash map works on the principle of hashing

HashMap get(Key k) method calls hashCode method on the key object and applies returned hashValue to its own static hash function to find a bucket location(backing array) where keys and values are stored in form of a **nested class called Entry (Map.Entry)** . So you have concluded that from the previous line that **Both key and value is stored in the bucket as a form of Entry object** . So thinking that Only value is stored in the bucket is not correct and will not give a good impression on the interviewer .

* Whenever we call get(Key k) method on the HashMap object . First it checks that whether key is null or not . Note that **there can only be one null key in HashMap .**

If key is null , then Null keys always map to hash 0, thus index 0.

If key is not null then , it will call hashfunction on the key object , see line 4 in above method i.e. key.hashCode() ,so after key.hashCode() returns hashValue , line 4 looks like

```
4.            int hash = hash(hashValue)
```

, and now ,it applies returned hashValue into its own hashing function .

We might wonder why we are calculating the hashvalue again using hash(hashValue). Answer is ,It defends against poor quality hash functions.

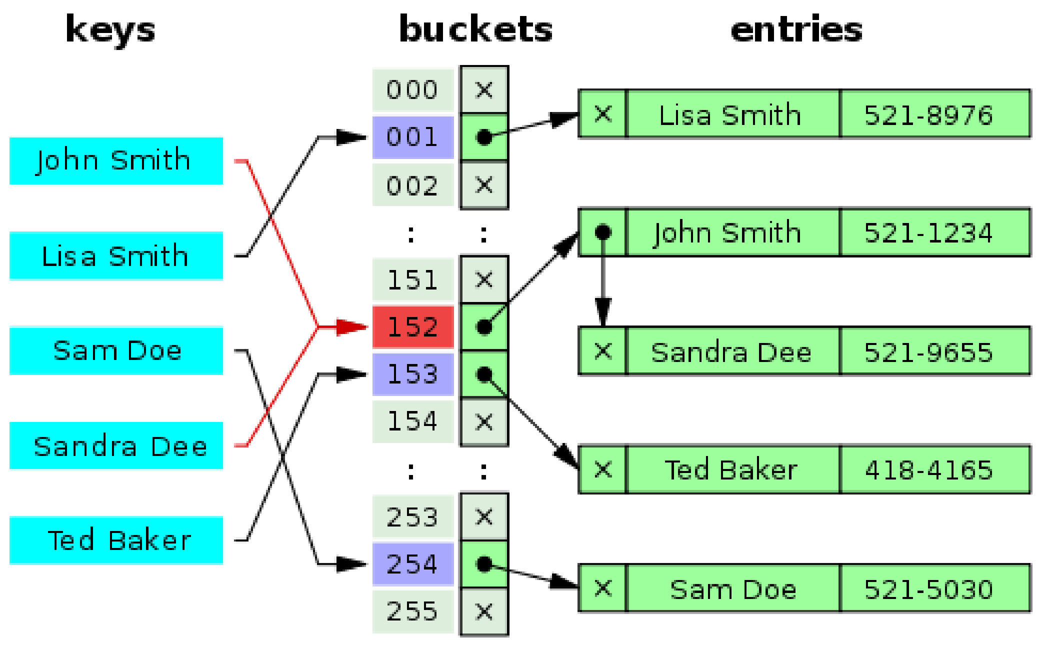
Now step 4 final hashvalue is used to find the bucket location at which the Entry object is stored . **Entry object stores in the bucket like this (hash,key,value,bucketindex) .**

Interviewer: What if when two different keys have the same hashcode ?

Solution, equals() method comes to rescue.Here candidate gets puzzled. Since bucket is one and we have two objects with the same hashcode .Candidate usually forgets that bucket is a simple linked list.

The bucket is the linked list effectively . Its not a LinkedList as in a java.util.LinkedList - It's a separate (simpler) implementation just for the map .

So we traverse through linked list , comparing keys in each entries using keys.equals() until it return true. Then the corresponding entry object Value is returned .



One of our readers Jammy asked a very good question

When the functions 'equals' traverses through the linked list does it traverses from start to end one by one...in other words brute method. Or the linked list is sorted based on key and then it traverses?

Answer is when an element is added/retrieved, same procedure follows:

a. Using key.hashCode() [see above step 4],determine initial hashvalue for the key

b. Pass intial hashvalue as hashValue in hash(hashValue) function, to calculate the final hashvalue.

c. Final hash value is then passed as a first parameter in the indexFor(int ,int)method .
The second parameter is length which is a constant in HashMap Java Api , represented by DEFAULT_INITIAL_CAPACITY

The default value of DEFAULT_INITIAL_CAPACITY is 16 in HashMap Java Api .

indexFor(int,int) method returns the first entry in the appropriate bucket. The linked list in the bucket is then iterated over - (the end is found and the element is added or the key is matched and the value is returned)

Explanation about indexFor(int,int) is below :

```
/**
```

```
* Returns index for hash code h.
*/
static int indexFor(int h, int length) {
    return h & (length-1);
}
```

The above function indexFor() works because Java HashMaps always have a capacity, i.e. number of buckets, as a power of 2.
 Let's work with a capacity of 256,which is 0x100, but it could work with any power of 2. Subtracting 1 from a power of 2 yields the exact bit mask needed to bitwise-and with the hash to get the proper bucket index, of range 0 to length - 1.
256 - 1 = 255
0x100 - 0x1 = 0xFF
E.g. a hash of 257 (0x101) gets bitwise-anded with 0xFF to yield a bucket number of 1.

Interviewer: What if when two keys are same and have the same hashcode ?
If key needs to be inserted and already inserted hashkey's hashcodes are same, and keys are also same(via reference or using equals() method) then override the previous key value pair with the current key value pair.

The other **important point** to note is that **in Map ,Any class(String etc.) can serve as a key if and only if it overrides the equals() and hashCode() method .**

Interviewer: How will you measure the performance of HashMap?

According to Oracle Java docs,

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(HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.), 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.

In HashMap class, the default value of load factor is (.75) .

Interviewer : What is the time complexity of Hashmap get() and put() method ?

The hashmap implementation provides constant time performance for (get and put) basic operations i.e the complexity of get() and put() is O(1) , assuming the hash function disperses the elements properly among the buckets.