1. **How does java Hashmap work internally**

Posted by [Dinesh Rajput](https://plus.google.com/117630637804698317816)

***What is Hashing?***  
Hashing in its simplest form, is a way to assigning a unique code for any variable/object after applying any formula/algorithm on its properties. A true Hashing function must follow this rule:  
  
Hash function should return the same hash code each and every time, when function is applied on same or equal objects. In other words, two equal objects must produce same hash code consistently.  
  
***Note:*** All objects in java inherit a default implementation of **hashCode()** function defined in Object class. This function produce hash code by typically converting the internal address of the object into an integer, thus producing different hash codes for all different objects.  
  
**HashMap is an array of Entry objects:**  
Consider HashMap as just an array of objects.  
  
Have a look what this Object is:

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

final K key;

V value;

Entry<K,V> next;

final int hash;

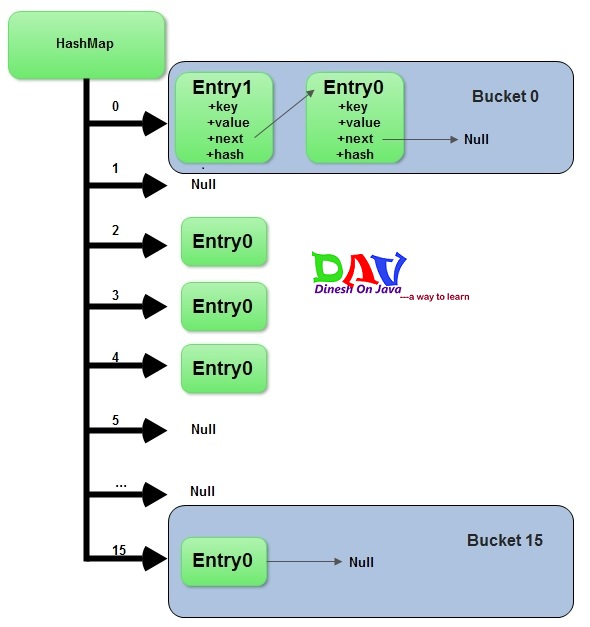
...

}

Each Entry object represents key-value pair. Field next refers to other Entry object if a bucket has more than 1 Entry.  
  
Sometimes it might happen that ***hashCodes*** for 2 different objects are the same. In this case 2 objects will be saved in one bucket and will be presented as ***LinkedList***. The entry point is more recently added object. This object refers to other object with next field and so one. Last entry refers to null.  
When you create ***HashMap*** with default constructor

HashMap hashMap = new HashMap();

Array is gets created with size 16 and default 0.75 load balance.



**Adding a new key-value pair**

1. Calculate ***hashcode*** for the key
2. Calculate position hash % (arrayLength-1)) where element should be placed(bucket number)
3. If you try to add a value with a key which has already been saved in ***HashMap***, then value gets overwritten.
4. Otherwise element is added to the bucket. If bucket has already at least one element - a new one is gets added and placed in the first position in the bucket. Its next field refers to the old element.

**Deletion:**

1. Calculate ***hashcode*** for the given key
2. Calculate bucket number (hash % (arrayLength-1))
3. Get a reference to the first Entry object in the bucket and by means of equals method iterate over all entries in the given bucket. Eventually we will find correct Entry. If desired element is not found - return null

**What *put()* method actually does:**  
Before going into **put()** method’s implementation, it is very important to learn that instances of ***Entry*** class are stored in an array. ***HashMap*** class defines this variable as:

/\*\*

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

\*/

transient Entry[] table;

Now look at code implementation of ***put()*** method:

/\*\*

\* Associates the specified value with the specified key in this map. If the

\* map previously contained a mapping for the key, the old value is

\* replaced.

\*

\* @param key

\* key with which the specified value is to be associated

\* @param value

\* value to be associated with the specified key

\* @return the previous value associated with <tt>key</tt>, or <tt>null</tt>

\* if there was no mapping for <tt>key</tt>. (A <tt>null</tt> return

\* can also indicate that the map previously associated

\* <tt>null</tt> with <tt>key</tt>.)

\*/

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;

}

**Lets note down the steps one by one:**  
  
**Step1-** First of all, key object is checked for null. If key is null, value is stored in table[0] position. Because hash code for null is always 0.  
  
**Step2**- Then on next step, a hash value is calculated using key’s hash code by calling its **hashCode()** method. This hash value is used to calculate index in array for storing Entry object. JDK designers well assumed that there might be some poorly written ***hashCode()*** functions that can return very high or low hash code value. To solve this issue, they introduced another hash() function, and passed the object’s hash code to this hash() function to bring hash value in range of array index size.  
  
**Step3-** Now ***indexFor(hash, table.length)*** function is called to calculate exact index position for storing the Entry object.  
  
**Step4**- Here comes the main part. Now, as we know that two unequal objects can have same hash code value, how two different objects will be stored in same array location [called bucket].  
  
Answer is **LinkedList**. If you remember, Entry class had an attribute “next”. This attribute always points to next object in chain. This is exactly the behavior of LinkedList.  
  
So, in case of collision, Entry objects are stored in LinkedList form. When an Entry object needs to be stored in particular index, HashMap checks whether there is already an entry?? If there is no entry already present, Entry object is stored in this location.  
  
If there is already an object sitting on calculated index, its next attribute is checked. If it is null, and current Entry object becomes next node in LinkedList. If next variable is not null, procedure is followed until next is evaluated as null.  
  
What if we add the another value object with same key as entered before. Logically, it should replace the old value. How it is done? Well, after determining the index position of Entry object, while iterating over LinkedList on calculated index, HashMap calls equals method on key object for each Entry object. All these Entry objects in LinkedList will have similar hash code but equals() method will test for true equality. If key.equals(k) will be true then both keys are treated as same key object. This will cause the replacing of value object inside Entry object only.  
  
In this way, HashMap ensure the uniqueness of keys.  
  
**How *get()* methods works internally**  
Now we have got the idea, how key-value pairs are stored in HashMap. Next big question is : what happens when an object is passed in get method of HashMap? How the value object is determined?  
  
Answer we already should know that the way key uniqueness is determined in put() method , same logic is applied in get() method also. The moment HashMap identify exact match for the key object passed as argument, it simply returns the value object stored in current Entry object.  
  
If no match is found, get() method returns null.  
  
Let have a look at code:

/\*\*

\* Returns the value to which the specified key is mapped, or {@code null}

\* if this map contains no mapping for the key.

\*

\* <p>

\* More formally, if this map contains a mapping from a key {@code k} to a

\* value {@code v} such that {@code (key==null ? k==null :

\* key.equals(k))}, then this method returns {@code v}; otherwise it returns

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

\*

\* </p><p>

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

\* the map contains no mapping for the key; it's also possible that the map

\* explicitly maps the key to {@code null}. The {@link #containsKey

\* containsKey} operation may be used to distinguish these two cases.

\*

\* @see #put(Object, Object)

\*/

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;

}