How a HashMap works internally has become a popular question in almost all interviews. Almost everybody knows how to use a HashMap or the difference between HashMap and Hashtable. However, many people fail when the question is "How does a HashMap work internally?"

The answer to this 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? 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. 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. Let's dig into that and see how it works.

How Does the Put() Method Work Internally?

The code 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 — if two different objects have the same hashcode (e.g. Aa and BB will have the same hashcode), will it be stored in the same bucket? To handle this, let's think of the LinkedList in the 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 the next attribute is not null, then it keeps the loop running until the 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 `{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, it will replace the value object inside the Entry class and not the key. This way, it prevents the duplicate key from being inserted.

How Does the Get() Method Work Internally?

Almost the same logic 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;
 }
```

1. First, it gets the hash code of the key object, which is passed, and finds the bucket location.

2. If the correct bucket is found, it returns the value.

3. 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.

I hope this article clarifies the troublesome HashMap internal mechanism. Happy learning!