COLLECTIONS HOW IT WORKS INTERNALLY:

HASHMAP:

**Internal Working of HashMap in Java**

In this article, we will see how the hashmap get and put method works internally. What operations are performed? How the hashing is done. How the value is fetched by key. How the key-value pair is stored.  
In the [previous article](https://www.geeksforgeeks.org/java-util-hashmap-in-java), HashMap contains an array of Node and Node can represent a class having the following objects :

1. int hash
2. K key
3. V value
4. Node next

Now we will see how this works. First, we will see the hashing process.

Hashing

Hashing is a process of converting an object into integer form by using the method hashCode(). It’s necessary to write the hashCode() method properly for better performance of HashMap. Here I am taking the key of my class so that I can override the hashCode() method to show different scenarios. My Key class is

//custom Key class to override hashCode()  
// and equals() method  
class Key  
{  
 String key;  
 Key(String key)  
 {  
 this.key = key;  
 }  
   
 @Override  
 public int hashCode()   
 {  
 return (int)key.charAt(0);  
 }  
  
 @Override  
 public boolean equals(Object obj)  
 {  
 return key.equals((String)obj);  
 }  
}

Here override hashCode() method returns the first character’s ASCII value as hash code. So whenever the first character of the key is same, the hash code will be the same. You should not approach these criteria in your program. It is just for demo purposes. As HashMap also allows a null key, so hash code of null will always be 0.

[hashCode() method](https://www.geeksforgeeks.org/equals-hashcode-methods-java/): hashCode() method is used to get the hash code of an object. hashCode() method of the object class returns the memory reference of an object in integer form. Definition of hashCode() method is public native hashCode(). It indicates the implementation of hashCode() is native because there is not any direct method in java to fetch the reference of the object. It is possible to provide your implementation of hashCode().   
In HashMap, hashCode() is used to calculate the bucket and therefore calculate the index.

[equals() method](https://www.geeksforgeeks.org/equals-hashcode-methods-java/): This method is used to check whether 2 objects are equal or not. This method is provided by the Object class. You can override this in your class to provide your implementation.

HashMap uses equals() to compare the key to whether they are equal or not. If the equals() method return true, they are equal otherwise not equal.

Buckets: A bucket is an element of the HashMap array. It is used to store nodes. Two or more nodes can have the same bucket. In that case, a link list structure is used to connect the nodes. Buckets are different in capacity. A relation between bucket and capacity is as follows:

capacity = number of buckets \* load factor

A single bucket can have more than one node, it depends on the hashCode() method. The better your hashCode() method is, the better your buckets will be utilized.

Index Calculation in Hashmap

Generated hash code may be in the range of integer and if we create arrays for such a range, then it will easily cause outOfMemoryException. So we generate an index to minimize the size of the array. The following operation is performed to calculate the index.

**index = hashCode(key) & (n-1).**

where n is the number of buckets or the size of the array. In our example, I will consider n as the default size which is 16.

**Why the above method is used to calculate the index**

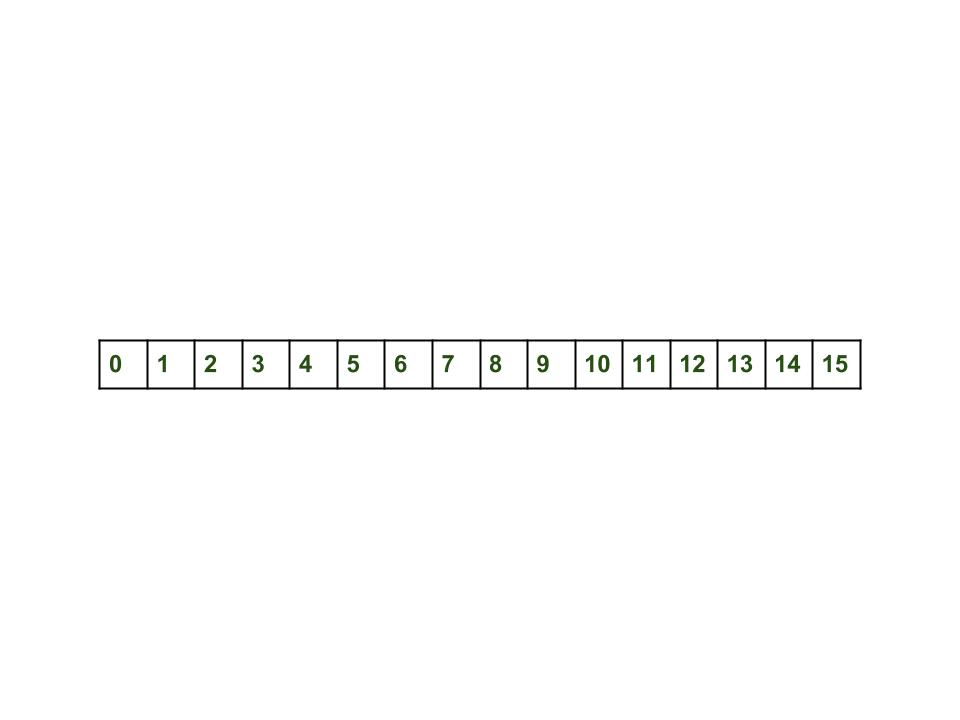
**Why the above method is used to calculate the index**

Using a bitwise AND operator is similar to doing bit masking wherein only the lower bits of the hash integer is considered which in turn provides a very efficient method of calculating the modulus based on the length of the hashmap.

* **Initially Empty hashMap:** Here, the hashmap’s size is taken as 16.

HashMap map = new HashMap();

* HashMap :



* **Inserting Key-Value Pair:** Putting one key-value pair in the above HashMap

map.put(new Key("vishal"), 20);

* **Steps:**
  1. Calculate hash code of Key {“vishal”}. It will be generated as 118.
  2. Calculate index by using index method it will be 6.
  3. Create a node object as :
* {  
   int hash = 118  
    
   // {"vishal"} is not a string but   
   // an object of class Key  
   Key key = {"vishal"}  
    
   Integer value = 20  
   Node next = null  
  }

Place this object at index 6, if no other object is presented there.

* **Inserting another Key-Value Pair:** Now, putting the other pair that is,

map.put(new Key("sachin"), 30);

* **Steps:**
  1. Calculate hashCode of Key {“sachin”}. It will be generated as 115.
  2. Calculate index by using index method it will be 3.
  3. Create a node object as :

{  
 int hash = 115  
 Key key = {"sachin"}  
 Integer value = 30  
 Node next = null  
}

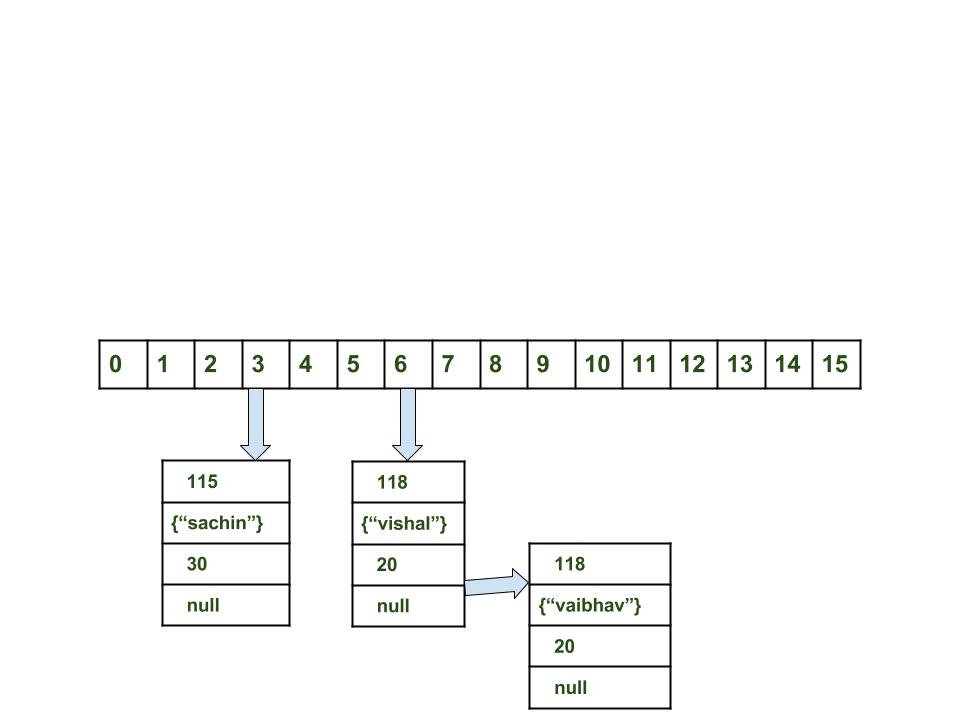
* **In Case of collision:** Now, putting another pair that is,

map.put(new Key("vaibhav"), 40);

* **Steps:**
  1. Calculate hash code of Key {“vaibhav”}. It will be generated as 118.
  2. Calculate index by using index method it will be 6.
  3. Create a node object as :

{  
 int hash = 118  
 Key key = {"vaibhav"}  
 Integer value = 40  
 Node next = null  
}

1. Place this object at index 6 if no other object is presented there.
2. In this case, a node object is found at index 6 – this is a case of collision.
3. In that case, check via the hashCode() and equals() method if both the keys are the same.
4. If keys are the same, replace the value with the current value.
5. Otherwise, connect this node object to the previous node object via linked list and both are stored at index 6.   
   Now HashMap becomes :



* **Fetch the data for key sachin:**

map.get(new Key("sachin"));

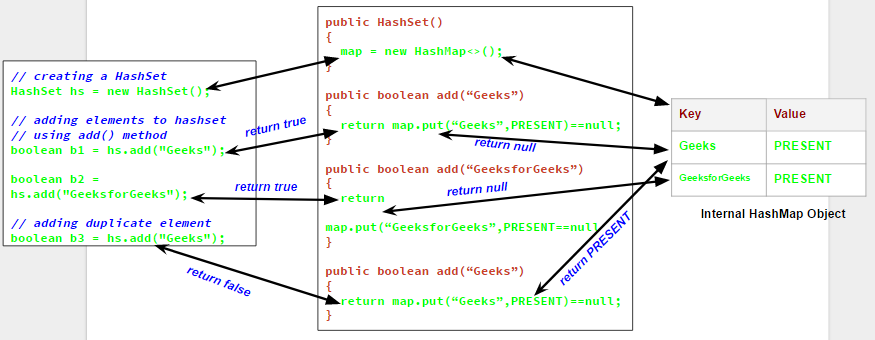
* **Steps:**
  1. Calculate hash code of Key {“sachin”}. It will be generated as 115.
  2. Calculate index by using index method it will be 3.
  3. Go to index 3 of the array and compare the first element’s key with the given key. If both are equals then return the value, otherwise, check for the next element if it exists.
  4. In our case, it is found as the first element and the returned value is 30.
* Fetch the data for key vaibhav:

map.get(new Key("vaibhav"));

* **Steps:**
  1. Calculate hash code of Key {“vaibhav”}. It will be generated as 118.
  2. Calculate index by using index method it will be 6.
  3. Go to index 6 of the array and compare the first element’s key with the given key. If both are equals then return the value, otherwise, check for the next element if it exists.
  4. In our case, it is not found as the first element and the next node object is not null.
  5. If the next node is null then return null.
  6. If the next of node is not null traverse to the second element and repeat process 3 until the key is not found or next is not null.
  7. Time complexity is almost constant for the put and the get method until rehashing is not done.
  8. In case of collision, i.e. index of two or more nodes are the same, nodes are joined by a link list i.e. the second node is referenced by the first node and the third by the second, and so on.
  9. If the key given already exist in HashMap, the value is replaced with the new value.
  10. hash code of the null key is 0.
  11. When getting an object with its key, the linked list is traversed until the key matches or null is found on the next field.

# Internal working of Set/HashSet in Java

As we know that a set is a **well-defined** collection of distinct objects. Each member of a set is called an element of the set. So in other words, we can say that a **set will never contain duplicate elements**. But how in java [Set](https://www.geeksforgeeks.org/set-in-java/) interface implemented classes like [HashSet](https://www.geeksforgeeks.org/hashset-in-java/), [LinkedHashSet](https://www.geeksforgeeks.org/linkedhashset-class-in-java-with-examples/), [TreeSet](https://www.geeksforgeeks.org/treeset-class-java-examples/) etc. achieve this uniqueness. In this post, we will discuss the hidden truth behind this uniqueness.



// predefined HashSet class

public class HashSet

{

// A HashMap object

private transient HashMap map;

// A Dummy value(PRESENT) to associate with an Object in the Map

private static final Object PRESENT = new Object();

// default constructor of HashSet class

// It creates a HashMap by calling

// default constructor of HashMap class

public HashSet() {

map = new HashMap<>();

}

// add method

// it calls put() method on map object

// and then compares it's return value with null

public boolean add(E e) {

return map.put(e, PRESENT)==null;

}

// Other methods in Hash Set

}

We will understand this with an example.Let us see the output of the following program which try to add duplicate elements in a HashSet.

|  |
| --- |
| // Java program to demonstrate  // internal working of HashSet  import java.util.HashSet;  class Test  {      public static void main(String args[])      {          // creating a HashSet          HashSet hs = new HashSet();          // adding elements to hashset          // using add() method          boolean b1 = hs.add("Great");          boolean b2 = hs.add("GreatLearning");            // adding duplicate element          boolean b3 = hs.add("Great");            // printing b1, b2, b3          System.out.println("b1 = "+b1);          System.out.println("b2 = "+b2);          System.out.println("b3 = "+b3);            // printing all elements of hashset          System.out.println(hs);        }  } |

it is clear that when we try to add a duplicate element to a set using *add()* method, it returns *false*, and element is not added to hashset, as it is already present. Now the question comes, how *add()* method checks whether the set already contains the specified element or not. It will be more clear if we have a closer look on the *add()* method and default constructor in HashSet class.

// predefined HashSet class

public class HashSet

{

// A HashMap object

private transient HashMap map;

// A Dummy value(PRESENT) to associate with an Object in the Map

private static final Object PRESENT = new Object();

// default constructor of HashSet class

// It creates a HashMap by calling

// default constructor of HashMap class

public HashSet() {

map = new HashMap<>();

}

// add method

// it calls put() method on map object

// and then compares it's return value with null

public boolean add(E e) {

return map.put(e, PRESENT)==null;

}

// Other methods in Hash Set

}

Now as you can see that whenever we create a HashSet, it internally creates a [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) and if we insert an element into this HashSet using add() method, it actually call put() method on internally created HashMap object with element you have specified as it’s key and constant Object called “PRESENT” as it’s value. So we can say that a Set achieves uniqueness internally through HashMap. Now the whole story comes around [how a HashMap and put() method internally works](https://www.geeksforgeeks.org/internal-working-of-hashmap-java/).

As we know in a [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) each key is unique and when we call put(Key, Value) method, it returns the previous value associated with key, or null if there was no mapping for key. So in add() method we check the return value of map.put(key, value) method with null value.

1. If map.put(key, value) returns null, then the statement “map.put(e, PRESENT) == null” will return true and element is added to the HashSet(internally HashMap).
2. If map.put(key, value) returns old value of the key, then the statement “map.put(e, PRESENT) == null” will return false and element is not added to the HashSet(internally HashMap).