**All core concepts of HashMaps in Java . The only article you need!**

Diagram

Description automatically generated

**Topics to cover :**

1. *What are HashMaps*
2. *Hashing*
3. *HashMap Internals*
4. *hashCode()*
5. *equals()*
6. *What if we only overwrite equals() and not hashCode() ?*
7. What if we only override hashCode() and not equals() ?
8. *Capacity*
9. *Collisions*
10. *Load factor*
11. *Rehashing*
12. [*How does put method actually work(Internal library code walkthrough)*](https://vivek-sinless.medium.com/internal-working-of-get-key-and-put-key-value-in-hashmaps-e114cac3d7dc)
13. [*How does get method actually work(Internal library code walkthrough)*](https://vivek-sinless.medium.com/internal-working-of-get-key-and-put-key-value-in-hashmaps-e114cac3d7dc)
14. [*Implementation of hashmaps when we want the key to be a custom class*](https://vivek-sinless.medium.com/create-a-custom-key-object-in-hashmaps-java-cf4d3ddbb377)

Topics 12, 13, and 14 are covered in separate articles.

**1. What are HashMaps ?**

HashMap class belongs to Java Collections framework and provides an implementation of Map interface. It stores data as Key/Value pairs. To access or get the data we pass the key to get the value.

The best case for insertion and lookup takes O(1) time. However the worst case time complexity is O(n).

Had we used list instead of Maps, the time complexity would almost never be O(1) as in this case list traversal is needed to find the right index while with HashMaps we use hashing to find the right bucket.

HashMaps achieve time complexity of O(1) because they use a concept called Hashing.

**2. Hashing**

We use indexes to access an array. Say we wanted to get the 2nd element from the array(arr), we get the value by running “arr[0] “. If you wanted the nth element you just access it by “arr[n-1]”

Arrays take O(1) time to access/lookup an element. Array uses index to get the data, the same way HashMap uses hash value to get the data in O(1)

Hash is an integer value generated by a hashing algorithm — a value which acts as an address inside the map to store key/value pairs. So when you try to lookup a specific key, hash is generated for that key, using hashCode() and the value is fetched in O(1) time because hash acts as your memory address itself. Remember changing the size of the HashMap will change the hash values for all keys by using a concept called rehashing.

**Remember this rule of hashing :**

Whenever it is invoked on the same object more than once during an execution of a Java application, *hashCode()* must consistently return the same value, provided no information used in equals comparisons on the object is modified. This value doesn’t need to stay consistent from one execution of an application to another execution of the same application.

**3. HashMaps Internals**

HashMap stores data in key/value pairs. It uses an algorithm called hashing to find the memory location or the bucket where it should be stored.

hashmap.put(1, “value1”) // 1 -> value1  
hashmap.put(2, ”value2”) // 2 -> value2  
hashmap.put(1, ”value3”) // 1 -> value3 (Notice the same keys ?)

Notice the same keys above ?

We are trying to insert an element again with the key value “1”. In this case when hash value is computed, it returns a hash value where key/value already exists, so this will just replace the value i.e. key1 will point to value3

To get the value we use map.get(keyName) :

hashmap.get(1) // outputs value3

HashMaps memory are divided into different buckets, these buckets are identified by hash value. It could happen that two keys can have the same hash value although the keys might be different. In this case we create a linked list in that bucket to store additional elements. That is the reason the worst case time complexity of hashmaps is O(n) .

*So although finding a hash took O(1) time, finding the key in the bucket takes O(n) time because internally every bucket creates a linked list to store additional elements with the same hash.  
To avoid this issue of O(n) time complexity, Java 8 and later versions use a balanced tree (also called a*[*red-black tree*](https://www.baeldung.com/cs/red-black-trees)*) instead of a*Linked List*to store collided entries. This improves the worst-case performance of*HashMap*from*O(n)*to*O(log n)*.*

**To store element HashMaps actually uses an array of objects with the structure :**

Graphical user interface, text, application

Description automatically generated

To create and maintain a hashmap we use a class which has hash, key, value, and next as the variables. Key and Value field is self explanatory, hash is the bucket address, next is to store the address to the next Node in the same bucket when two keys although different have same hash.

Diagram

Description automatically generated

The above image is taken from geeksforgeeks.

**4. hashcode()**

This method computes the integer hash value.

We can write our own custom logic to generate a hashcode. We just need to override the hashcode method.

@Override  
**public int** hashCode() {  
 // write your custom logic  
}

An example with custom logic:

@Override  
public int **hashCode**() { return (int) id \* name.hashCode() \* email.hashCode();}

Hash is exactly i the integer value obtained after performing bitwise AND operation on the value of hash of the key and array size minus one.



[We will discuss more about this in our next article of how a get and put works.](https://vivek-sinless.medium.com/internal-working-of-get-key-and-put-key-value-in-hashmaps-e114cac3d7dc)

Two keys can have the same hash value even if they have different memory references considering you have overridden the hashCode() with your logic.

**5. equals()**

This method is used to check if two key objects are equal. If two key objects are equal, it would point to the same bucket. Hence any put request on a map with the same key and same hash will overwrite the older value.

It could also happen that two different keys can have the same bucket address based on the hashCode algorithm that you might be using. In such a scenario, a linked list is created on the bucket to store additional values.

We overwrite equals() and hashCode() often. We overwrite it when we want the keys for hashmaps to be our own objects instead of an Integer or String.

[Custom objects as key.](https://vivek-sinless.medium.com/create-a-custom-key-object-in-hashmaps-java-cf4d3ddbb377)

Remember equals() and hashcode() go hand in hand when you define Key as your custom object.

**6. What if we only overwrite equals() and not hashCode() ?**

If you do not overwrite a hashCode and use the default implementation provided by the internal library, then every time a new hash value will be generated even if the key objects are equal according to equals(). In this case even if your custom equals() tells you that the two objects in comparison are equal, the key/value pairs won’t be overwritten. The hash value generated will be different, so the two objects with same keys will be stored in separate buckets instead of the older one being overwritten.

**7. What if we only override hashCode() and not equals() ?**

Say that you insert a key/value in the bucket. After that if you try to insert the same key(with different memory reference but same content)with a different value, ideally it should be overwritten. However when you do not overwrite equals(), the two key objects can not figure out that they are the same object in fact based on the same content. Hence instead of overwriting, it adds one more element to the linkedlist for that bucket.

**8. Capacity**

Every key/value pair is stored in buckets in a hashmap. Capacity is the number of buckets present in a hashmap. The default capacity of a HashMap is 16. As the number of key/value pairs increases, the chances of a collision in a hashmap increases, hence the bucket capacity should increase using load factor and rehashing.

This is how you define the capacity manually :

Map<String, String> mapWithInitialCapacity = new HashMap<>(5);

**9. Collisions**

If the *hashCode()* method is well-written, *HashMap* will distribute the items across all the buckets. Therefore, *HashMap* stores and retrieves entries in constant time *O(1)*. However many times the key computes to the same hash, which increases the complexity to maintain elements in the same bucket.

Collisions happen when different key objects have the same hash values. The more collisions you have, the more linked lists will be created for every bucket. If this continues happening, the complexity will increase and the insertion/lookup starts taking more time .

If you remember, the whole objective of this data structure was to compute insertion and lookup in a constant time. But collisions act as the instigator to diverge from the only objective of this data structure. The solution to this is the Load Factor.

**10. Load Factor**

This value helps decide when to increase the capacity of a hashmap. If we have more space in a hashmap, we have more buckets, which means we can generate more hash values which will eventually result in fewer chances of collision.

The default capacity of a hashmap is 16. If we have 16 nodes in this hashmap — one node for every bucket, the time taken for lookup will be O(1). If we have 32 nodes then every bucket will take 2 nodes each meaning the lookup will traverse 2 elements. If we have 64, every bucket will have 4 nodes each in the linked list, the time complexity will take 4 lookups i.e a traversal of 4 elements.

Load Factor by default is 75% of the capacity of a hashmap. Whenever the hashmap is 75% filled with key/value pairs, the hashmaps size will be doubled by rehashing.

This is how we can change the default value of a load factor.  
The creation of a hashmap takes 2 parameters here : 1. Capacity 2. Load factor

Map<String, String> mapWithInitialCapacityAndLF = new HashMap<>(5, 0.5f);

A higher value of load factor decreases the space overhead but increases the lookup cost. The low value of load factor will decrease the chances of collisions but will start taking up more space.

**11. Rehashing**

Rehashing is the process of re-calculating the hash values of already stored entries. When the load factor increases(default value of load factor is 0.75), the complexity increases, so to overcome this, size is doubled using rehashing whenever threshold of 0.75 is reached.