Internal working of HashMap

We have learned **HashMap in java** in recent posts. As we know HashMap works based on  
key-value pair that is known as the **entry**. But how does HashMap internal implementation works? It is one of the most common questions from **HashMap interview questions**. In this post, we will see the **internal working of HashMap in java**or **HashMap internal implementation in java.**

**Here is the table content of the article will we will cover this topic.  
1. [HashMap internal implementation or Internal working of HashMap?](https://javagoal.com/internal-working-of-hashmap/" \l "1)  
2.**[**What is hashing in java?**](https://javagoal.com/internal-working-of-hashmap/#2) **3.**[**How does put method work?**](https://javagoal.com/internal-working-of-hashmap/#3) **4.**[**How collisions are resolved?**](https://javagoal.com/internal-working-of-hashmap/#4) **5.**[**How does get method work?**](https://javagoal.com/internal-working-of-hashmap/#5)

Before moving further, we should know about the **equals() method**and**hashCode() method.** **HashMap** stores the data in Key-Value pair and it uses node to store the key-value. It uses the **Node class** that is a static inner class to represent the pair(Key-Value) as a node.  The **HashMap** contains an array and LinkedList data structure internally for storing Key and Value. Let’s see the object of the structure of HashMap or **Internal working of HashMap**in JDK:

final int hash;

final K key;

V value;

Node<K,V> next;

In HashMap, each node has some specific data that is **hashCode**, **key**, **value,**and **address** of the next node.

**HashMap internal implementation** **or Internal working of HashMap**

The internal implementation of HashMap depends upon the hash code of the object.  It performs each operation based on **hashCode() method** and **equals() method**. If you are not familiar with the **hashcode() method** and **equals() method,** then please read it from here.  
The **HashMap** uses the **Hashing technique** to store and retrieve the data. Hashing is a technique that is used to get the unique integer value.

**What is hashing in java?**

**Hashing** is a technique that converts an object into a unique integer value. This unique integer number is known as the hashcode.  The hashing technique uses the hash function, which invokes the **hashcode() method** to generate a number.

Whenever we put any object in **HashMap**, the **put(key, value) method**calculates the **hashCode**of the key and inserts the value in **HashTable**. But this is not a blind process, it takes a few steps to put the object in **HashTable**. To understand the process, we need to understand **how does put(key, value) method work?**

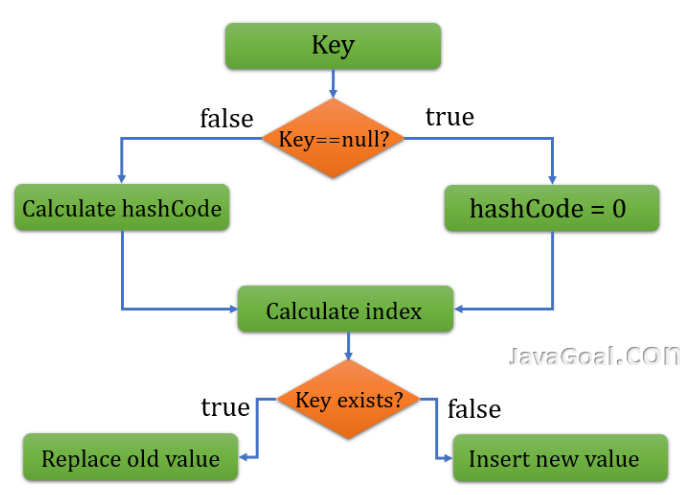
**How does put method work?**

1. First of all, it checks the key object is null or not? If the given key is null, then it will directly go for position 1 i.e table[0]. Because the **hashcode**for null is always 0.

2. If the key is not null then the **hashCode() method** is used to get the hash value. The hash value is used to calculate the index value in the table to store the key-value pair.

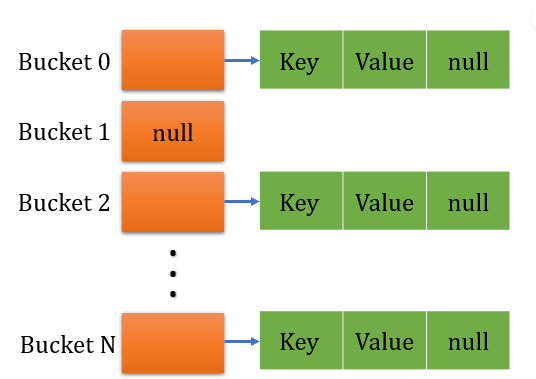
3. There might be a chance, the hashCode() method can return a very high or low hash code value. It totally depends on the implementation of the **hashCode() method** in the class. To solve this issue, the **hash() function** to bring hash value in the range of array index size.

4. After calculating the hash code, it calls the **indexFor(hash, table.length) method** to calculate the particular index to store the pair. It accepts the hash code of the key and length of the *table[]* array. And returns the index value.

5.  Each pair stores as a node on a specific index in the table and that called a bucket. A bucket is used to stores the nodes. A bucket can contain two or more nodes and the LinkedList structure is used to connect the nodes. We will discuss it later in the **Hash collision**. 

Now you have understood how a HashMap stores all the values in **HashTable** based on a unique key.  It replaces the value if the given key already exists in **HashMap**.

Let’s take an example and see how it works in memory. As you can see, there can be any number of buckets in HashMap. Each bucket maintains the LinkedList structure of nodes.



**How collisions are resolved?**

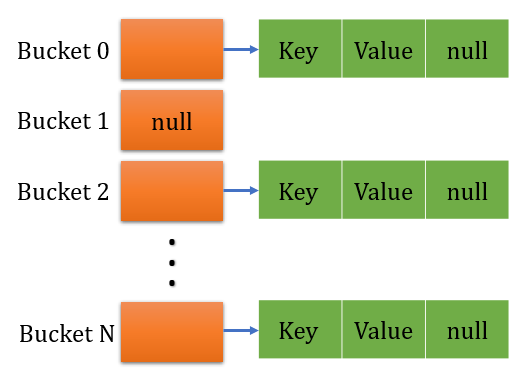
We have seen how **HashMap**stores each pair by use of hashcode. But there may be a chance that two unequal objects can have the same hash code value? Suppose two objects have the same hashcode then how it stores the object in **HashTable**? Let’s find the answer

If there are two different objects having the same hashcode then they will be stored in the same bucket. The bucket uses the LinkedList structure to store more than two nodes. Here HashMap performs a few steps.

1.**HashMap** checks, whether there is an entry already exist with the same in hashcode in a bucket or not?

2. If there is no entry already present, then node is stored at the index value of table.

3. But if entry is already existing then it checks the reference of the node. If the node is not referencing any next node then-current entry object becomes the next node in LinkedList. But If it is referencing, then the procedure is followed until next is evaluated as null.



**How does get method work?**

We have seen **HashMap** stores the data in buckets and there can be any number of buckets. Each bucket stores multiple nodes in the **LinkedList structure**. But one question comes in mind, how does **get(key) method** work based on key. The get(key) method performs some steps.

Step 1: Like **put() method**, it also checks whether the given key is null or not. If the given key is null, it calls the getForNullKey() method.

Step 2 : If the given key is not null, then it internally calculates the hash code of the specified key is calculated.

Step 3 : Get the index value by use of *indexFor()* method. It uses the hashcode value.

Step 4 : By use of index value it gets the node from bucket that contains key value pair. If bucket has multiple nodes in LinkedList structure, it will iterate though linked list at that position and checks for the key using equals() method.

<https://medium.com/interviewnoodle/how-does-hashmap-works-internally-619debad797f>

# How HashMap Works Internally In Java?

## What is a hashmap?

If you have ever used any form of cache, you have used hashmap. Even distributed caching systems like Couchbase, Redis, etc are sort of hashmaps extended to handle high concurrency and large-scale data through replication and distribution.

In Java, if you want to represent a group of objects as key-value pairs then you can go for the **Map interface**. Each key-value pair is called an “**Entry**”. So, Map is considered as a collection of entry objects. **HashMap<K, V>**which provides a basic implementation of the Map interface and is available under java.util package is one of the most efficient/optimal data-structure for storing data/objects.

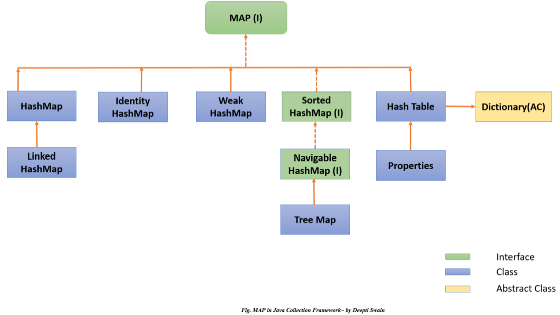


Fig 1. Map Interface in Java Collection Framework ~ by

[Deepti Swain](https://medium.com/u/f3e9d7075028?source=post_page-----619debad797f--------------------------------)

**Map Interface Methods:** There are various methods as shown below are available in the map interface, which is implemented by HashMap. We will focus on put() and get() methods.

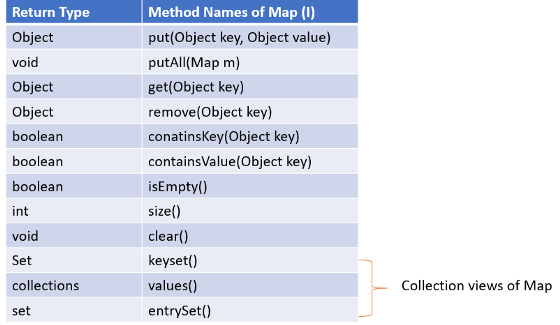


Fig 2. Map/HashMap supported Methods~ by

[Deepti Swain](https://medium.com/u/f3e9d7075028?source=post_page-----619debad797f--------------------------------)

**put(Object key, Object value):** put method used for inserting/storing the objects in to map by using the hashcode of the key.

**get(Object key):**get method used for searching/retrieving/getting the value using the hashcode of the key.

**HashMap Behaviour:**

* Duplicate values are allowed in HashMap, whereas duplicate keys are not allowed, if you try to insert the duplicate keys, then it replaces the corresponding key’s old value with the new value.

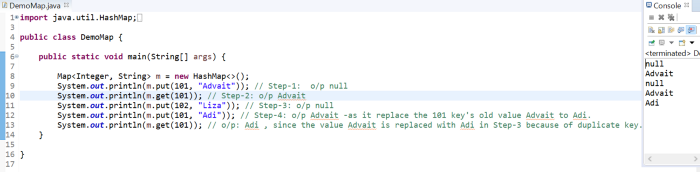


Fig 3. Duplicate Keys Not Preferable in HashMap ~ by

[Deepti Swain](https://medium.com/u/f3e9d7075028?source=post_page-----619debad797f--------------------------------)

* Example: In the above program, for key “101”, in step-1 “Advait” is inserted, but in step-3 for the same key “101”, inserted value is “Adi”. HashMap replaced the “Advait” value with “Adi” and shared the older value in step-3. In step 4, while retrieving the value using m.get(Key) method, we got the latest value available i.e. Adi.
* Heterogeneous objects are allowed for both key and value. Null is allowed for the key for only one time. Null is allowed for values for any number of times.
* Hash Map implements a Serializable and cloneable interface but not random access.
* The underline data structure of HashMap is Hash Table. Insertion order is not preserved and it is based on the hashcode of the keys.

Now, let’s focus on how does this hashmap works internally which makes it a more efficient data structure for inserting and searching data.

# ****How does hashmap work internally?****

To understand HashMap internal working we need to know about a few major topics such as buckets, hashing, hashCode(), equals(), initial capacity, load factor, etc. So first let’s focus on those.

***1) What is bucket in hashMap?***



Fig 4. Representation of a HashMap Node ~ by

[Deepti Swain](https://medium.com/u/f3e9d7075028?source=post_page-----619debad797f--------------------------------)

HashMap contains an array of Nodes, which is called a bucket and each node is represented as a class having the four objects i.e int hash, K key, V value, Node next.

***2) What is hash/hashing?***

In general, hashing is a process of converting an bigger value into a smaller value. In Java hashing converts the object into an integer form by using the method hashCode(). It's necessary to write the hashCode() method properly for better performance of HashMap.

***3) What is hashCode()?***

HashMap works on the basis of hashing which uses the hashCode() from Object class. You can [override](https://deeptiswain.medium.com/oops-concept-in-java-96d500c86e97) this hashCode() from Object class and write your own implementation on how hashing value you want to set for the Objects. hashCode() returns an integer value, generated by a hashing algorithm is used for deciding which bucket (the index)an object should be placed into.

//custom hashCode()method could be:  
**@Override public int hashCode(int key) {   
return key%m;** // larger integers will be divided by a small number m to get a smaller object/value.  
}

A good hashCode() should have:

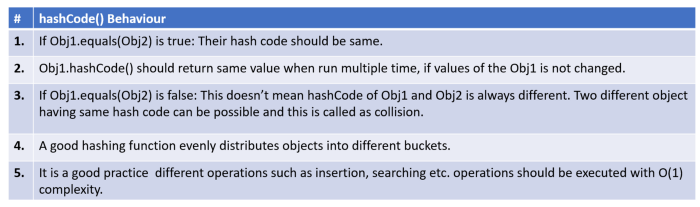


Fig 5. Good Practice for hashing/hashCode() ~by

[Deepti Swain](https://medium.com/u/f3e9d7075028?source=post_page-----619debad797f--------------------------------)

***4) What is equals()?***

equals() used to compare the content/value of two objects. If two objects are not from the same class immediately false is return. Then it checks if value/ contents of objects are same or not, if not same returns false, else true.HashMap uses equals() to compare the key whether they are equal or not. If equals() method return true, they are equal otherwise not equal.

There are two important factors that affect the performance of a hashmap. That is the initial capacity and load factor.

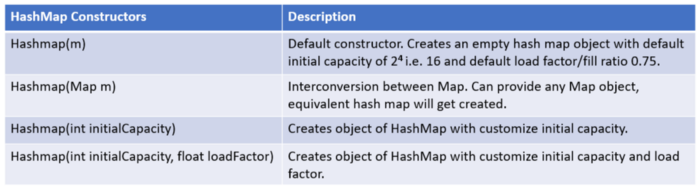


Fig 6. HashMap Constructors ~ by

[Deepti Swain](https://medium.com/u/f3e9d7075028?source=post_page-----619debad797f--------------------------------)

* We have to choose these two factors very carefully while creating the HashMap object as these impact the insertion (i.e put()) and retrieval(i.e. get()) [complexity](https://medium.com/interviewnoodle/time-complexity-of-a-program-3a5f80be23dc)of the operation.

***5) What is Initial Capacity?***

When we create an object of hashmap, it creates a bucket with an initial default size of 2⁴, i.e., 16. As the capacity of HashMap reaches the threshold, it gets doubled in the form of 2^n each time, such as 2⁵ i.e. the bucket size becomes 32**,**2⁶ i.e. the bucket size becomes 64 and so on.

***6) What is Load Factor?***

The Load factor is a measure that decides when to increase the HashMap capacity to maintain the insertion (i.e put()) and retrieval(i.e. get()) operation complexity of O(1). The default load factor of HashMap is 0.75f (75% of the map size).

***7) When to increase the HashMap bucket size/capacity of hashMap?***

There are two way to determine when to increase the HashMap bucket size.

**Threshold = initial capacity \* Load factor**

1. The initial capacity of hashmap is =16, The default load factor of hashmap=0.75. So when 16\*0.75 =12. So, 12th index key-value pair of hashmap will keep its size to 16. As soon as 13th element (key-value pair) will come into the HashMap, it will increase its size from default 16 buckets to 32 buckets following 2^n.

**m/n > 0.75** // increases the HashMap bucket size/capacity.  
where m = number of entries in a hashmap  
n = total size of hashmap/bucket size

2. After inserting the first element by checking the hash code of key, it checks whether increase of the hashmap capacity required or not by using the formula m/n.

**1/16 = 0.0625** — — — — — — → 0.0625 < 0.75 **// No need to increase the hashmap size.2/16 = 0.125** — — — — — — → 0.125 < 0.75 **// No need to increase the hashmap size.**  
.  
.  
.  
**12/16 = 0.75** — — — — — — — -> 0.75 = 0.75 **// No need to increase the hashmap size.  
13/16 = 0.8125** — — — — — — → 0.8125 > 0.75 **//Need to increase the hashmap size.**

To keep put() and get() complexity O(1), it is advisable to have a load factor around 0.75.

Now that we are aware of the important terminologies, let’s focus how the insertion and search operation takes place in HashMap.

1. **Insertion/ put():**

Step 1: Creating Bucket:

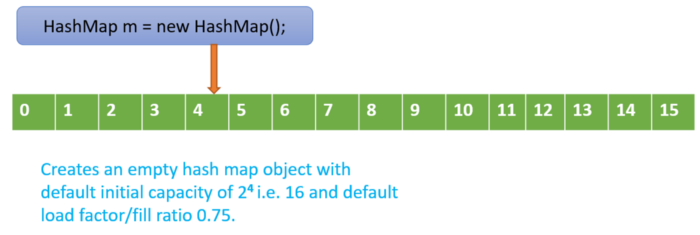


Fig 7: Step 1: HashMap internal working: Creating Bucket ~

[Deepti Swain](https://medium.com/u/f3e9d7075028?source=post_page-----619debad797f--------------------------------)

Step 2: Finding the hashcode and index value:

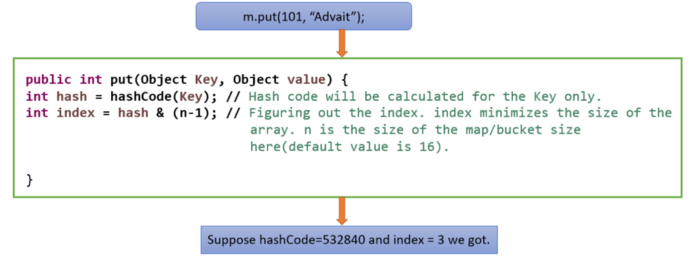
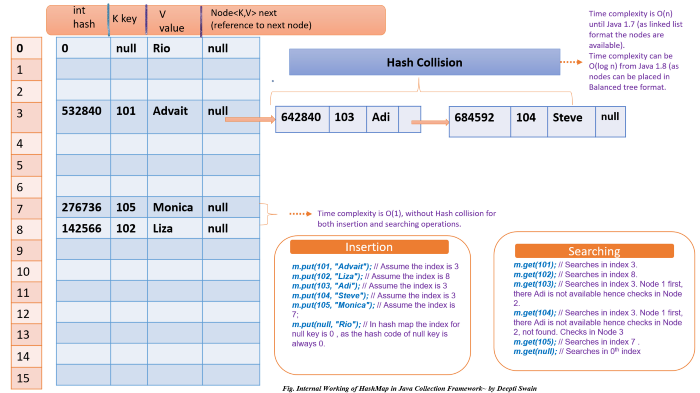


Fig 8. Step 2: Finding the hashcode and index value ~ by

[Deepti Swain](https://medium.com/u/f3e9d7075028?source=post_page-----619debad797f--------------------------------)

Step 3: Insert the objects into the corresponding index of the bucket.



**Fig 9. Internal Working of HashMap in Java Collection Framework~ by Deepti Swain**

* hashCode() for the key gets generated and using that the index of the first key 101 fetched. Assume the index value is 3. Then [JVM](https://medium.com/interviewnoodle/jvm-architecture-71fd37e7826e) will fill the corresponding nodes with hash, key, value and next node reference as null.
* Similarly for 102 key, the index is 8. JVM will fill the 8th index node with 4 details.
* For 103 key, the index is 3. It checks if in 3rd index, any entry is already available. If yes, then it creates another node, and old 101’s next node reference points to this new 103 node instead of null. Same for the case 104 key. These 3 nodes(101,103,104) gives a view of linked list. When the calculated index value is same for two or more Keys, multiple nodes gets created at the single index, this is called as **hash collision**.
* As the index of null key is always 0, hence it placed at 0th index.
* In the best case the complexity of inserting items into HashMap is O(1). In the worst case the [time complexity](https://medium.com/interviewnoodle/time-complexity-of-a-program-3a5f80be23dc) for insertion operation/put() is 0(n) till jdk1.8. To improve this, in jdk 1.8, when number of entries/nodes become larger than a certain threshold, the hash will change from using a linkedlist to a balance tree which will improve the worst case performance from O(n) to O(log n).

2.**Searching/ Retriving/ get():**

Step 1: Figure out the hashcode of the key.

Step 2: Fetch the index value using the hashcode.

Step 3: In the HashMap bucket, at the corresponding index, check if the hashcode is matching. Check if the key is matching using the equals().

Step 4: If it matches, then it fetches and returns the corresponding value.

Step 5: If not matched, it keeps on looking at next nodes of the same index, until the next node reference comes as null.

Step 6: The time complexity is the same as insertion operation.

HashMap is the best choice if our frequent operation is search/insertion operation.