There are four things we should know about HashMap before going into the internals of how does **HashMap** work in Java.

1. [Hashing](https://codepumpkin.com/hashmap-internal/#Hashing)
2. [Map.Entry Interface / Node Class](https://codepumpkin.com/hashmap-internal/#EntryObj)
3. [hashCode() Method](https://codepumpkin.com/hashmap-internal/#HashCode)
4. [equals() Method](https://codepumpkin.com/hashmap-internal/#Equals)

**1. Hashing**

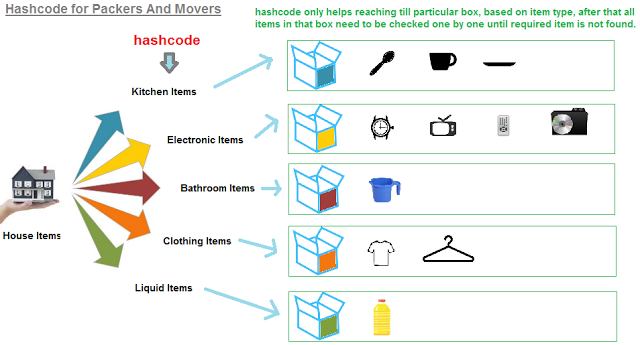
**HashMap** works on the principal of hashing.

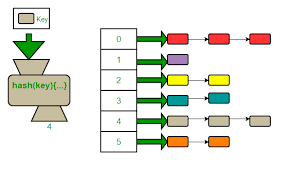
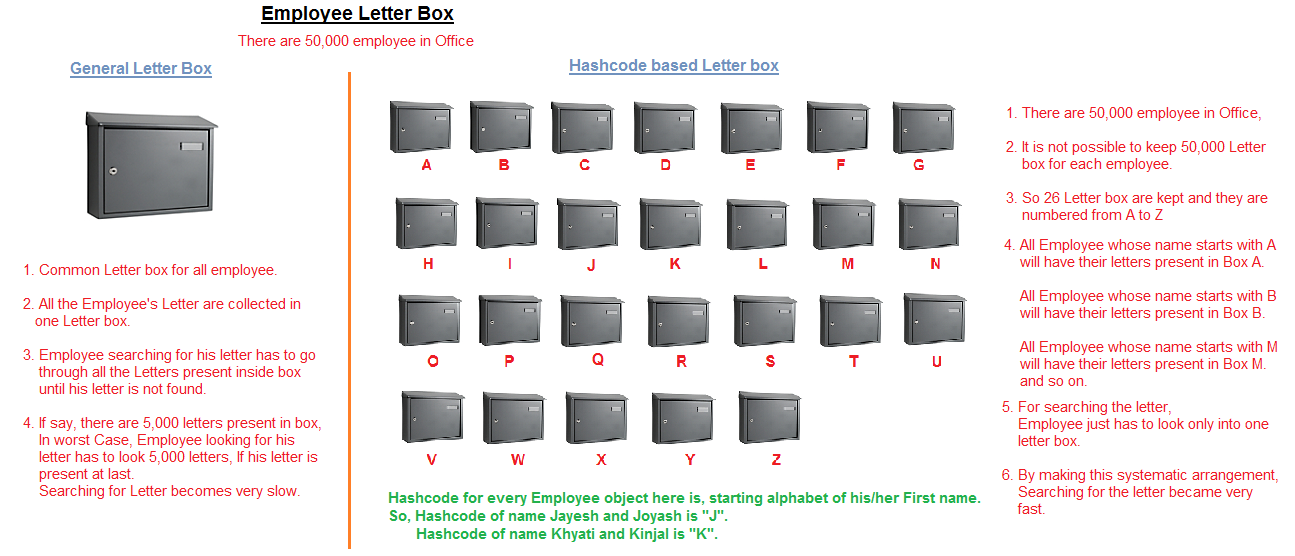
You want to store n number of addresses. Instead of maintaining one address on each page(Huge no of pages ) You will use address book

 (Address Book) **Hashing**is the process of indexing and retrieving element (data) in a data structure to provide faster way of finding the element using the **hash key.** ( Letters A,B,C)

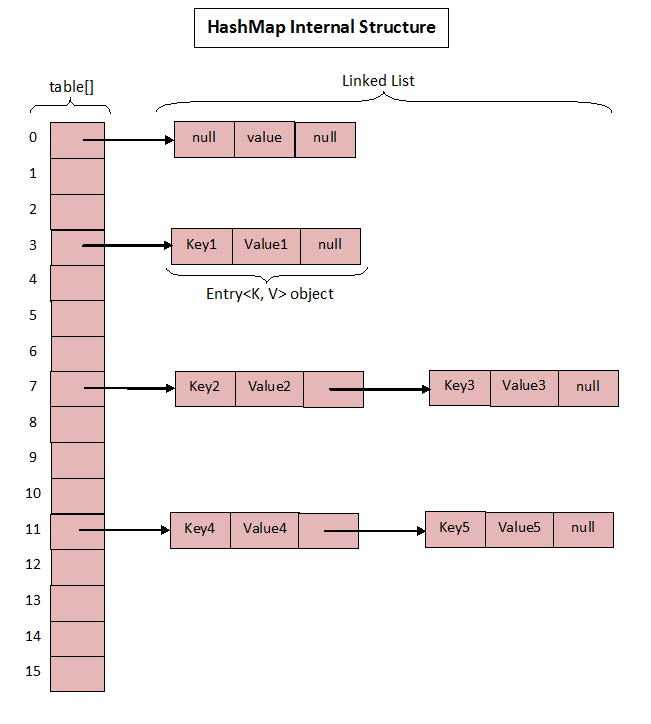
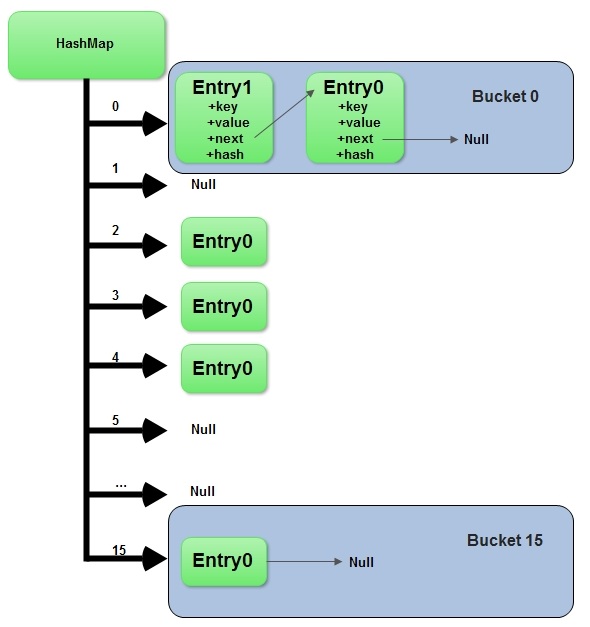
**Hash codes** are typically used to increase the performance of large collections of data.

Collections such as HashMap and HashSet use the hash code value of an object to determine **where the object should be stored** in the collection, and the hashcode is used again **to help locate the object** in the collection.





**2. Map.Entry Interface (jdk 1.7)/ Node Class(jdk 1.8)**

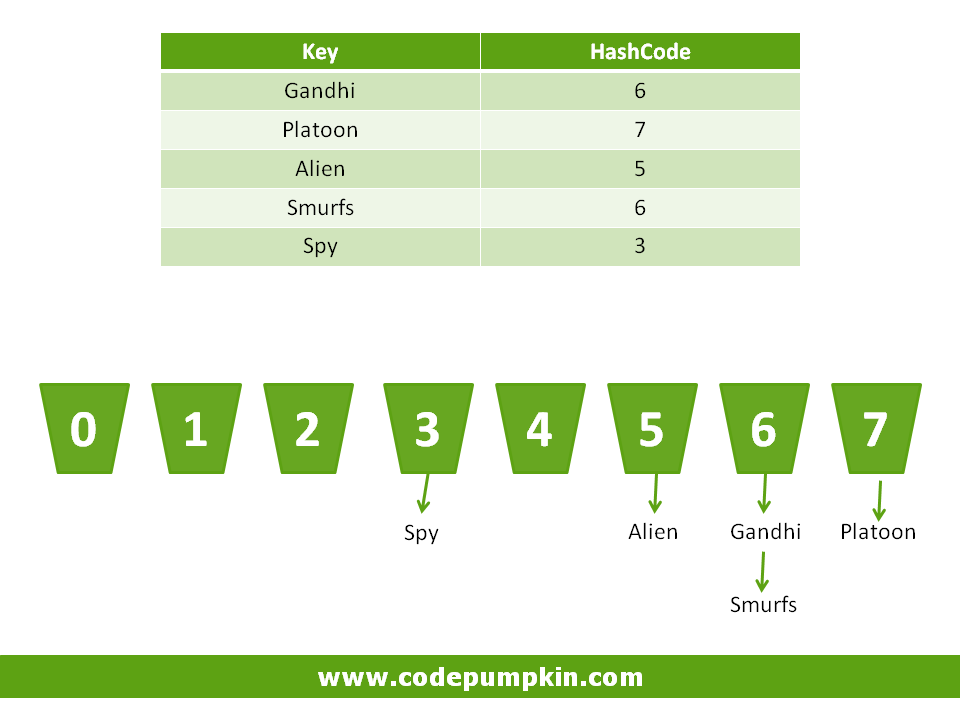
HashMap maintains an **array of buckets.** Each bucket is a linked list of a private static inner class called an Entry Class which encapsulates the key-value pairs.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | static class Entry<K,V> implements Map.Entry<K,V> {          final int hash;          final K key;          V value;          Node<K,V> next;         // constructor and methods  } |

Bucket term used here is actually an index of array, that array is called table in HashMap implementation. Thus table[0] is referred as bucket0, table[1] as bucket1 and so on.

|  |  |
| --- | --- |
| 1  2  3  4  5 | /\*\*     \*  The table, resized as necessary. Length MUST Always be a power of two.     \*\*/        transient Entry[] table; |

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101



**3. HashCode() Method**

The value returned by hashCode() is the object's hash code, which is the object's memory address in hexadecimal.

Hashcode is a unique code which is generated by the JVM for every object creation.

hashCode() is used for *bucketing* in Hash implementations like HashMap, HashTable, HashSet, etc.

The value received from hashCode() is used as the *bucket number* for storing elements of the set/map. This bucket number is the *address* of the element inside the set/map.

When you do contains() it will take the hash code of the element, then look for the bucket where hash code points to. If more than 1 element is found in the same bucket (multiple objects can have the same hash code), then it uses the equals() method to evaluate if the objects are equal, and then decide if contains() is true or false, or decide if element could be added in the set or not.

Hash codes are typically used to increase the performance of large collections of data.

Collections such as HashMap and HashSet use the hash code value of an object to determine where the object should be stored in the collection, and the hashcode is used again to help locate the object in the collection. Let’s understand this by following example.

**Example:**

Imagine a set of buckets lined up on the floor. Someone hands you a piece of paper with movie name on it. You take the movie name and calculate an integer code (i.e. Hashcode) for each name by calculating total number of characters in that name.

Imagine that each bucket represents a different code number (Hashcode) and place the piece of paper into corresponding bucket according to its Hashcode value.

Here, movie **Gandhi** has 6 letters, so we will put it into the Bucket with value 6.  **Platoon**has 7 letters, so it will be placed into 7th bucket and so on.

Now imagine that someone comes up and shows you a name and says, "Please retrieve the piece of paper that matches this name." So you look at the name they show you, and run the same hashcode-generating algorithm. The hashcode tells you in which bucket you should look to find the name.

**4. Equals() Method**

In above example, you might have noticed that two different names may result in the same value. Such situation is known as **Hashing Collision.**

For example, the movie names **Gandhi**and **Smurfs**have the same number of letters, so the hashcode will be identical for both names. That's acceptable, but it means that when someone asks you for the **Gandhi**piece of paper, you'll still have to search through the target bucket reading each name until we find **Gandhi**rather than **Smurfs**.

The hashcode tells you only about which bucket to go into, but not how to locate the name once we're in that bucket.

In such cases,**equals()** method is used to locate the exact match.

So for efficiency, your goal is to have the papers distributed as evenly as possible across all buckets. Ideally, you should have just one name per bucket so that when someone asks for a paper you can simply calculate the hashcode and just grab the one paper from the correct bucket (without having to go flipping through different papers in that bucket until you locate the exact one you're looking for).

The least efficient hashcode generator returns the same **hashcode**for all the movie names, so that all the papers lands in the same bucket while the others buckets remain empty.

In real-life hashing, it’s common to have more than one entry in a bucket. Hashing retrieval is a two-step process.

1. Find the right bucket (using hashCode())
2. Search the bucket for the right element (using equals() ).

HashMap provides **put(key, value)** for **storing** and **get(key)**method for **retrieving** Values from HashMap.

**How Put() Method Works Internally?**

1. Using hashCode() method, hash value will be calculated. Using that hash it will be ascertained, in which bucket particular entry will be stored.
2. equals() method is used to find if such a key already exists in that bucket, if no then a new node is created with the map entry and stored within the same bucket. A linked-list is used to store those nodes.
3. If equals() method returns true, which means that the key already exists in the bucket. In that case, the new value will overwrite the old value for the matched key.

**In Case Of Null Key**

As we know that HashMap also allows null, though there can only be one null key in HashMap. While storing the Entry object HashMap implementation checks if the key is null, in case key it is null, it always maps it to bucket 0 as hash is not calculated for null keys.

**How Get() Method Works Internally?**

1. Using the key again, hash value will be calculated to determine the bucket where that Entry object is stored.
2. In case there are more than one Entry objects are stored within the same bucket as a linked-list, equals() method will be used to find out the correct key.
3. As soon as the matching key is found, get() method will return the value object stored in the Entry object.

**HashMap Changes In Java 8**

Though HashMap implementation provides constant **time performance** **O(1)** for get() and put() method, but that is in the ideal case when the Hash function distributes the objects evenly among the buckets.

The performance may worsen in the case hashCode() is not proper and there are lots of hash collisions. As we know that in case of hash collision entry objects are stored as a node in a linked-list and equals() method is used to compare keys. That comparison to find the correct key within a linked-list is a linear operation so in a worst case scenario the **complexity**becomes **O(n)**.

To address this issue in Java 8 hash elements use balanced trees instead of linked lists after a certain threshold is reached. Which means HashMap starts with storing Entry objects in linked list, but after the number of items in a hash becomes larger than a certain threshold, the hash will change from using a linked list to a balanced tree. This improves the worst case performance from**O(n)** to **O(log n)**.