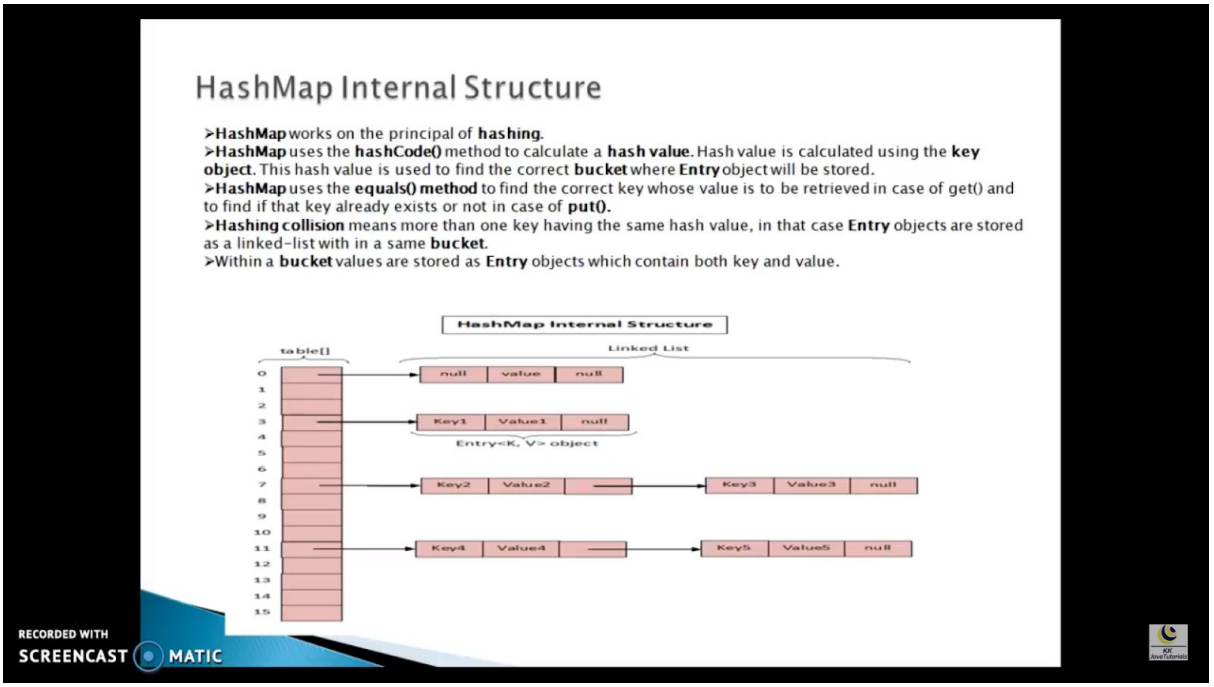
HashMap :



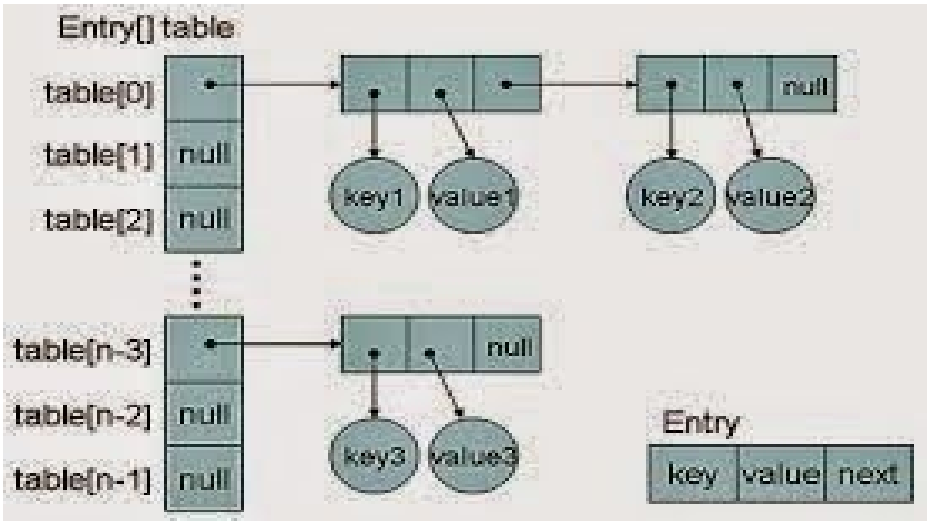
HashMap in Java works on hashing principle. It is a data structure which allows us to store object and retrieve it in constant time `O(1)` provided we know the key. In hashing, hash functions are used to link key and value in HashMap. Objects are stored by calling `put(key, value)` method of HashMap and retrieved by calling `get(key)` method. When we call put method, hashcode() method of the key object is called so that hash function of the map can find a bucket location to store value object, which is actually an index of the internal array, known as the table.

HashMap internally stores mapping in the form of **`Map.Entry`** object which contains both key and value object. When you want to retrieve the object, you call the get() method and again pass the key object. This time again key object generate same hash code (it's mandatory for it to do so to retrieve the object and that's why HashMap keys are immutable e.g. String) and we end up at same bucket location. If there is only one object then it is returned and that's your value object which you have stored earlier. Things get little tricky when collisions occur.

Since the internal array of HashMap is of fixed size, and if you keep storing objects, at some point of time hash function will return same bucket location for two different keys, this is called collision in HashMap. In this case, a linked list is formed at that bucket location and a new entry is stored as next node.

If we try to retrieve an object from this linked list, we need an extra check to search correct value, this is done by equals() method. Since each node contains an entry, HashMap keeps comparing entry's key object with the passed key using equals() and when it return true, Map returns the corresponding value.

Since searching in linked list is O(n) operation, in worst case hash collision reduce a map to linked list. This issue is recently addressed in Java 8 by replacing linked list to the tree to search in `O(logN)` time.



What happens On HashMap in Java if the size of the HashMap exceeds a given threshold defined by load factor ?

If the size of the Map exceeds a given threshold defined by load-factor e.g. if the load factor is `.75` it will act to re-size the map once it filled 75%. Similar to other collection classes like ArrayList, Java HashMap re-size itself by creating a new bucket array of size twice of the previous size of HashMap and then start putting every old element into that new bucket array. This process is called `rehashing` because it also applies the hash function to find new bucket location.

there is potential race condition exists while resizing `HashMap` in Java, if two thread at the same time found that now `HashMap` needs resizing and they both try to resizing. on the process of resizing of `HashMap` in Java, the element in the bucket which is stored in linked list get reversed in order during their migration to new bucket because Java HashMap doesn't append the new element at tail instead it append new element at the head *to avoid tail traversing*. If race condition happens then you will end up with an infinite loop.

1) Why String, Integer and other wrapper classes are considered good keys?
`String`, `Integer` and other wrapper classes are natural candidates of `HashMap` key, and `String` is most frequently used key as well because String is immutable and final, and overrides `equals` and `hashcode()` method. Other wrapper class

also shares similar property. Immutability is required, in order to prevent changes on fields used to calculate `hashCode()` because if key object returns different hashCode during insertion and retrieval than it won't be possible to get an object from `HashMap`.

Immutability is best as it offers other advantages as well like thread-safety, If you can keep your hashCode same by only making certain fields final, then you go for that as well.

Since `equals()` and `hashCode()` method is used during retrieval of value object from `HashMap`, it's important that key object correctly override these methods and follow contact. If unequal object returns different hashcode than chances of collision will be less which subsequently improve the performance of HashMap.

2) Can we use any custom object as a key in HashMap?
This is an extension of previous questions. Of course you can use any `Object` as key in Java `HashMap` provided it follows `equals and hashCode contract` and its hashCode should not vary once the object is inserted into Map. If the custom object is Immutable than this will be already taken care because you can not change it once created.

3) Can we use ConcurrentHashMap in place of Hashtable?
This is another question which getting popular due to increasing popularity of ConcurrentHashMap. Since we know Hashtable is synchronized but ConcurrentHashMap provides better concurrency by only locking portion of map determined by concurrency level. ConcurrentHashMap is certainly introduced as Hashtable and can be used in place of it, but Hashtable provides stronger thread-safety than ConcurrentHashMap.

What will happen if two different HashMap key objects have the same hashcode?
They will be stored in the same bucket but no next node of linked list. And keys equals () method will be used to identify correct key value pair in HashMap.

Java 8 improvement in Hashmap