How does HashMap work internally in java

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

1. Hashing
2. Map.Entry Interface / Node Class
3. hashCode() Method
4. equals() Method

1. Hashing

HashMap works on the principal of hashing. **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.

2. Map.Entry interface / Node class

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

```
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.

```
/**
   *  The table, resized as necessary. Length MUST Always be a power of two.
   **/

    transient Entry[] table;
```

3. hashCode() method

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

Collections such as HashMap and HashSet use the hashcode 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.

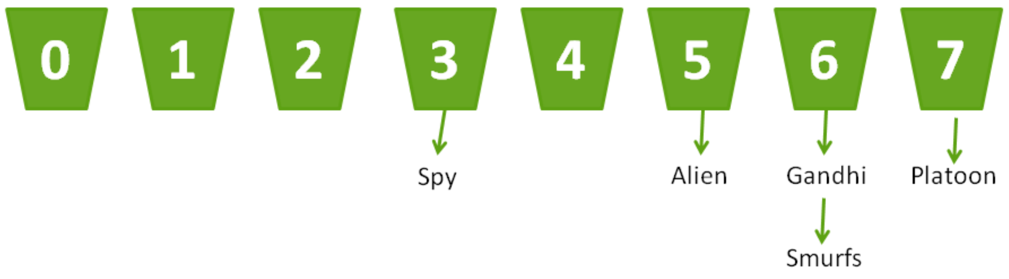
Example:

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.

Key	HashCode
Gandhi	6
Platoon	7
Alien	5
Smurfs	6
Spy	3



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)**.

Concurrent / Thread-Safe Maps

HashMap is not thread-safe. If multiple threads are accessing the same `HashMap` object and try to modify the structure of the `HashMap` (using `put()` or `remove()` method), it may cause an inconsistency in the state of `HashMap`.

To use `HashMap` in multithreaded environment, you must write your relevant code inside synchronized block or use any external Lock implementation. But in that case there are high chances of errors and deadlock situations, if proper care has not been taken.

In short, it is not advisable to use `HashMap` in multithreaded environment. Instead use any of the similar thread-safe collections like `Hashtable`, `Collections.SynchronizedMap` or `ConcurrentHashMap`.

Though all of them are thread-safe, `ConcurrentHashMap` provides better performance than remaining two. let's understand them one by one.

Hashtable

`Hashtable` is a legacy class available since jdk 1.1 which uses `synchronized` methods to achieve thread safety. At a time only one thread can read or write into `Hashtable`. In other word, thread acquires lock on entire `Hashtable` instance. Hence its performance is quite slow and we can not utilize the advantages of multithreaded architecture.

Another point to note about `Hashtable` is that it does not allow `null` keys or values whereas `HashMap` allows one `null` key and any number of `null` values.

Collections.SynchronizedMap

`SynchronizedMap` is a static inner class of utility class `java.util.Collections`. It is quite similar to `Hashtable` and it also acquires lock on entire Map instance. It is not a legacy class like `Hashtable` and it was introduced in jdk 1.5.

It provides functionality to convert any thread-unsafe Map implementation to thread-safe implementation using Collections.synchronizedMap(Map map) static method. For Example,

```
import java.util.*;
public class SynchronizedMapDemo {
   public static void main(String[] args) {
   // create HashMap
   Map<String,String> map = new HashMap<String,String>();

   // populate the map
   map.put("1","Malay");
   map.put("2","Ankit");
   map.put("3","Chintan");

   // create a synchronized map
   Map<String,String> syncmap = Collections.synchronizedMap(map);

   System.out.println("Synchronized map is : "+syncmap);
   }
}
```

Output

```
Synchronized map is : {3=Chintan, 2=Ankit, 1=Malay}
```

It internally wraps all the methods of Map interface with synchronized keyword. For example, here is the internal `put()`, `get()` and `remove()` method implementation of `SynchronizedMap` :

```
private static class SynchronizedMap<K,V>
        implements Map<K,V>, Serializable {

        //other instance variables
        private final Map < k ,v > m;     // Backing Map
        final Object      mutex;        // Object on which to
synchronize


        SynchronizedMap(Map<K,V> m) {
            if (m==null)
                throw new NullPointerException();
            this.m = m;
            mutex = this;
        }
```

```
            // Other constructors and methods

            public V get(Object key) {
                synchronized(mutex) {return m.get(key);}
            }

            public V put(K key, V value) {
                synchronized(mutex) {return m.put(key, value);}
            }
            public V remove(Object key) {
                synchronized(mutex) {return m.remove(key);}
            }
    }
```

Similar to `HashMap`, Synchronized `HashMap` allows one null key and multiple null values as it just wraps the methods of `HashMap` with synchronized keyword. Rest of the behavior remains same as original collection.

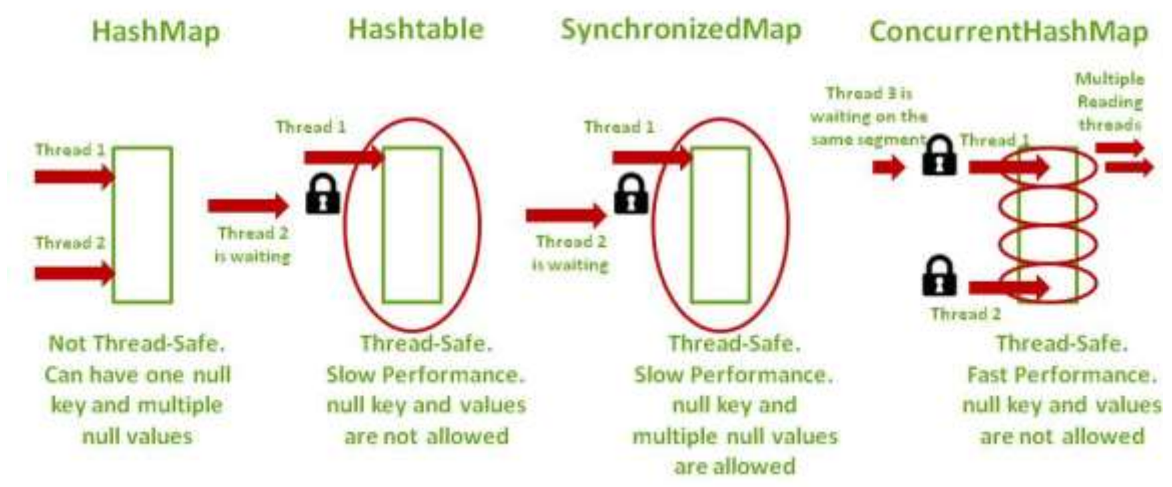
You can also check `SynchronizedSortedMap` which is similar data structure to convert thread-unsafe `TreeMap` and other `SortedMap` implementation to corresponding thread-safe collection.

ConcurrentHashMap

`Hashtable` and `SynchronizedMap` both acquires lock on entire Map object which provides thread-safety, but not good performance as at a time only one thread can access that Map instance.

To overcome this issue, `ConcurrentHashMap` was introduced in Java 5 along with other concurrent classes
like `CountDownLatch, CyclicBarrier`, `CopyOnWriteArrayList`, BlockingQueue within java.util.Concurrent package.

More than one threads can read and write concurrently in `ConcurrentHashMap` and still it provides thread-safety. Amazing, isn't it? How is it implemented internally?


HashMap
Hashtable
SynchronizedMap
ConcurrentHashMap
Thread 1
Thread 2
Not Thread-Safe. Can have one null key and multiple null values
Thread 1
Thread 2 is waiting
Thread-Safe. Slow Performance. null key and values are not allowed
Thread 1
Thread 2 is waiting
Thread-Safe. Slow Performance. null key and multiple null values are allowed
Thread 3 is waiting on the same segment
Thread 1
Multiple Reading threads
Thread 2
Thread-Safe. Fast Performance. null key and values are not allowed

Well, `ConcurrentHashMap` divides the Map instance into different segments. And each thread acquires lock on each segment. By default it allows 16 threads to access it simultaneously without any external synchronization i.e. by default concurrency level is 16. We can also increase or decrease the default concurrency level while creating `ConcurrentHashMap` by using below constructor :

```
ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel
)
```

Can multiple thread write in the same segment?

No. Thread acquires a lock on segment in `put ()` operation and at a time only one thread can write in that segment.

Can two threads write in the different segment?

Yes. Two threads are allowed to write concurrently in different segments.

Can multiple thread read from the same segment?

Yes. Thread doesn't acquire a lock on segment in `get ()` operation and any number of threads can read from the same segment.

If one thread is writing in a segment, can another thread read from that segment ()?

Yes. But in this case last updated value will be seen by the reading thread.

Null keys and values

`ConcurrentHashMap` doesn't allow `null` keys and `null` values.

Summary

1. `HashMap` is not thread-safe.
2. Once the size of `Hashtable` and `SynchronizedMap` becomes considerable large because for the iteration it has to be locked for the longer duration.
   While in `ConcurrentHashMap`, even if its size become very large, only portion or segment of the Map is locked which improves the performance in multithreading environment.
3. `SynchronizedMap` just wraps original Map object with `synchronized` block.
4. `Hashtable` and `ConcurrentHashMap` doesn't allow null keys and null values, whereas `SynchronizedMap` may allow `null` keys and `null` values based on the original collection class being passed inside it.