*HashMap* is the most used data structure in java because it gives almost constant time performance of O(1) for put and get operations irrespective of how big is the data. As you already know, *HashMap* stores the data in the form of key-value pairs. In this post, we will see how HashMap works internally in java and how it stores the elements to give O(1) performance for put and get operations.

HashMap Internal Structure :

*HashMap* stores the data in the form of key-value pairs. Each key-value pair is stored in an object of *Entry<K, V>* class. *Entry<K, V>* class is the static inner class of *HashMap* which is defined like below.

[?](http://javaconceptoftheday.com/how-hashmap-works-internally-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | static class Entry<K,V> implements Map.Entry<K,V>  {          final K key;          V value;          Entry<K,V> next;          int hash;            //Some methods are defined here  } |

As you see, this inner class has four fields. *key*, *value*, *next* and *hash*.

**key** : It stores the key of an element and its final.

**value** : It holds the value of an element.

**next** : It holds the pointer to next key-value pair. ***This attribute makes the key-value pairs stored as a linked list.***

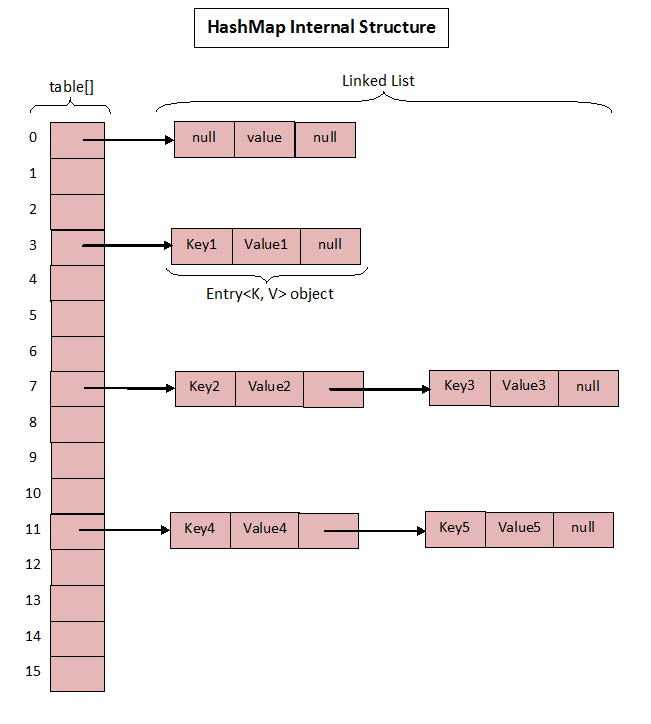
**hash** : It holds the hashcode of the key.

These *Entry* objects are stored in an array called *table[]*. This array is initially of size 16. It is defined like below.

[?](http://javaconceptoftheday.com/how-hashmap-works-internally-in-java/)

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

To summarize the whole *HashMap* structure, each key-value pair is stored in an object of *Entry<K, V>* class. This class has an attribute called *next* which holds the pointer to next key-value pair. This makes the key-value pairs stored as a linked list. All these *Entry<K, V>* objects are stored in an array called *table[]*. The below image best describes the *HashMap* structure.



The above image roughly shows how the *HashMap* stores its elements. Internally it uses an array of *Entry<K, V>* class called *table[]* to store the key-value pairs. But how *HashMap* allocates slot in *table[]* array to each of its key-value pair is very interesting. It doesn’t inserts the objects as you put them into *HashMap* i.e first element at index 0, second element at index 1 and so on. Instead it uses the hashcode of the key to decide the index for a particular key-value pair. It is called ***Hashing***.

What Is Hashing?

The whole *HashMap* data structure is based on the principle of **Hashing**. Hashing is nothing but the function or algorithm or method which when applied on any object/variable returns an unique integer value representing that object/variable. This unique integer value is called ***hash code***. Hash function or simply hash said to be the best if it returns the same hash code each time it is called on the same object. Two objects can have same hash code.

Whenever you insert new key-value pair using *put()* method, *HashMap* blindly doesn’t allocate slot in the *table[]*array. Instead it calls hash function on the key. *HashMap* has its own hash function to calculate the hash code of the key. This function is implemented so that it overcomes poorly implemented *hashCode()* methods. Below is implementation code of *hash()*.

[?](http://javaconceptoftheday.com/how-hashmap-works-internally-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | /\*\*       \* Retrieve object hash code and applies a supplemental hash function to the       \* result hash, which defends against poor quality hash functions.  This is       \* critical because HashMap uses power-of-two length hash tables, that       \* otherwise encounter collisions for hashCodes that do not differ       \* in lower bits. Note: Null keys always map to hash 0, thus index 0.       \*/      final int hash(Object k) {          int h = 0;          if (useAltHashing) {              if (k instanceof String) {                  return sun.misc.Hashing.stringHash32((String) k);              }              h = hashSeed;          }            h ^= k.hashCode();            // This function ensures that hashCodes that differ only by          // constant multiples at each bit position have a bounded          // number of collisions (approximately 8 at default load factor).          h ^= (h >>> 20) ^ (h >>> 12);          return h ^ (h >>> 7) ^ (h >>> 4);      } |

After calculating the hash code of the key, it calls *indexFor()* method by passing the hash code of the key and length of the *table[]* array. This method returns the index in the *table[]* array for that particular key-value pair.

[?](http://javaconceptoftheday.com/how-hashmap-works-internally-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | /\*\*       \* Returns index for hash code h.       \*/      static int indexFor(int h, int length) {          return h & (length-1);      } |

Now, let’s see how *put()* method works in detail.

How put() method works?

Below is the code implementation of *put()* method in the *HashMap* class.

[?](http://javaconceptoftheday.com/how-hashmap-works-internally-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | /\*\*       \* Associates the specified value with the specified key in this map.       \* If the map previously contained a mapping for the key, the old       \* value is replaced.       \*       \* @param key key with which the specified value is to be associated       \* @param value value to be associated with the specified key       \* @return the previous value associated with <tt>key</tt>, or       \*         <tt>null</tt> if there was no mapping for <tt>key</tt>.       \*         (A <tt>null</tt> return can also indicate that the map       \*         previously associated <tt>null</tt> with <tt>key</tt>.)       \*/      public V put(K key, V value) {          if (key == null)              return putForNullKey(value);          int hash = hash(key);          int i = indexFor(hash, table.length);          for (Entry<K,V> e = table[i]; e != null; e = e.next) {              Object k;              if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {                  V oldValue = e.value;                  e.value = value;                  e.recordAccess(this);                  return oldValue;              }          }            modCount++;          addEntry(hash, key, value, i);          return null;      } |

Let’s see how this code works step by step.

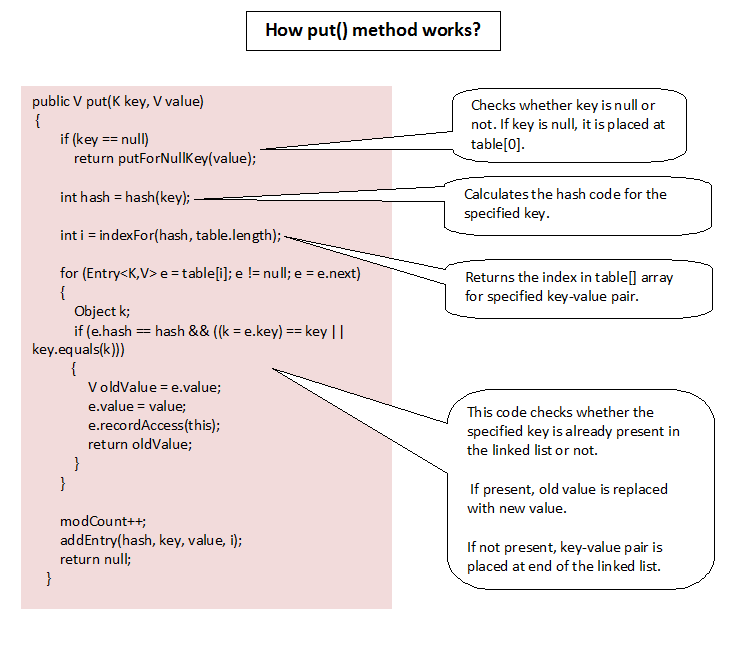
Step 1 : First checks whether the key is null or not. If the key is null, it calls *putForNullKey()* method. *table[0]* is always reserved for null key. Because, hash code of null is 0.

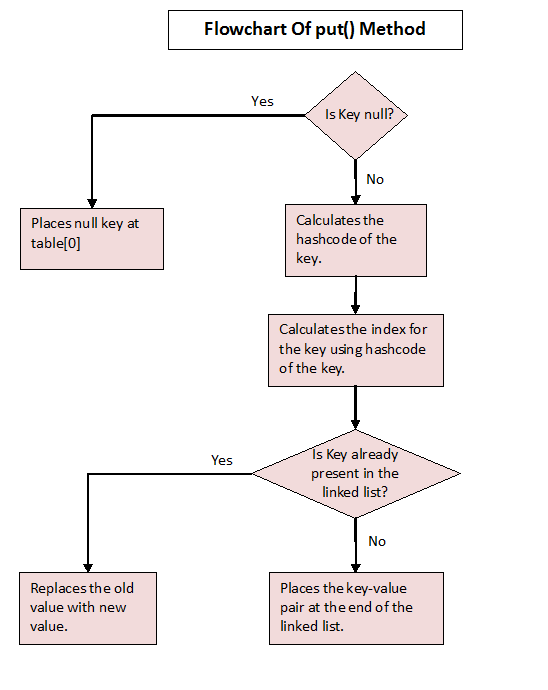
Step 2 : If the key is not null, then it calculates the hash code of the key by calling *hash()* method.

Step 3 : Calls *indexFor()* method by passing the hash code calculated in step 2 and length of the *table[]* array. This method returns index in *table[]* array for the specified key-value pair.

Step 4 : After getting the index, it checks all keys present in the linked list at that index ( or bucket). If the key is already present in the linked list, it replaces the old value with new value.

Step 5 : If the key is not present in the linked list, it appends the specified key-value pair at the end of the linked list.





How get() method Works?

Let’s see how get() method has implemented.

[?](http://javaconceptoftheday.com/how-hashmap-works-internally-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | /\*\*  \* Returns the value to which the specified key is mapped, or {@code null}  \* if this map contains no mapping for the key.  \*  \*    \* More formally, if this map contains a mapping from a key {@code k} to a  \* value {@code v} such that {@code (key==null ? k==null :  \* key.equals(k))}, then this method returns {@code v}; otherwise it returns  \* {@code null}. (There can be at most one such mapping.)  \*  \*    \* A return value of {@code null} does not <i>necessarily</i> indicate that  \* the map contains no mapping for the key; it's also possible that the map  \* explicitly maps the key to {@code null}. The {@link #containsKey  \* containsKey} operation may be used to distinguish these two cases.  \*  \* @see #put(Object, Object)  \*/  public V get(Object key) {      if (key == null)      return getForNullKey();      int hash = hash(key.hashCode());      for (Entry<K , V> e = table[indexFor(hash, table.length)]; e != null; e = e.next) {          Object k;          if (e.hash == hash && ((k = e.key) == key || key.equals(k)))              return e.value;      }      return null;  } |

Step 1 : First checks whether specified key is null or not. If the key is null, it calls *getForNullKey()* method.

Step 2 : If the key is not null, hash code of the specified key is calculated.

Step 3 : *indexFor()* method is used to find out the index of the specified key in the *table[]* array.

Step 4 : After getting index, it will iterate though linked list at that position and checks for the key using equals() method. If the key is found, it returns the value associated with it. otherwise returns null.