**The Map interface:**

The Map interface in java is one of the four top level interfaces of Java Collection Framework along with List, Set and Queue interfaces. But, unlike others, it doesn’t inherit from Collection interface. Instead, it starts it’s own interface hierarchy for maintaining the key-value associations. Map is an object of key-value pairs where each key is associated with a value. This interface is the replacement for ‘Dictionary‘class which is an abstract class introduced in JDK 1.0.

**HashMap**, **LinkedHashMap** and **TreeMap** are three popular implementations of Map interface.

**Properties Of Map Interface In Java :**

1) Map interface is a part of Java Collection Framework, but it doesn’t inherit **Collection Interface**.

2) Map interface stores the data as a **key-value pairs** where each key is associated with a value.

3) A map cannot have duplicate **keys** but can have duplicate **values**.

4) Each key **at most** must be associated with one value.

5) Each key-value pairs of the map are stored as **Map.Entry** objects. Map.Entry is an inner interface of Map interface.

6) The common implementations of Map interface are **HashMap**, **LinkedHashMap** and **TreeMap**.

7) Order of elements in map is implementation dependent. **HashMap** doesn’t maintain any order of elements. **LinkedHashMap** maintains **insertion order** of elements. Where as **TreeMap** places the elements according to supplied **Comparator**.

8) The Map interface provides three methods, which allows map’s contents to be viewed as a **set of keys**(keySet() method), **collection of values**(values() method), or **set of key-value mappings**(entrySet() method).

**Methods Of Map Interface In Java :**

|  |  |  |
| --- | --- | --- |
| SL NO. | Methods | Descriptions |
| 1 | int size() | Returns number of key-value pairs in this map. |
| 2 | boolean isEmpty() | Checks whether this map is empty or not. |
| 3 | boolean containsKey(Object key) | Returns true if this map contains a mapping for the specified key. |
| 4 | boolean containsValue(Object value) | Returns true if this map contains one or more keys associated with the specified value. |
| 5 | V get(Object key) | Returns value associated with the specified key. |
| 6 | V put(K key, V value) | Adds the specified key-value pair to this map. If the specified key already exist in the map, old value will be replaced by the specified value. |
| 7 | V remove(Object key) | Removes the specified key along with it’s value from this map. |
| 8 | void putAll(Map<? extends K, ? extends V> m) | Copies all key-value pairs from the specified map to this map. |
| 9 | void clear() | Removes all mappings from this map. |
| 10 | Set<K> keySet() | Returns a set containing all keys of this map. The returned set is backed by actual map. So, changes made to the map are reflected in the set and vice-versa. |
| 11 | Collection<V> values() | Returns a collection of values of this map. The returned collection is backed by actual map. So, any changes made to the map is reflected in collection and vice-versa. |
| 12 | Set<Map.Entry<K, V>> entrySet() | Returns set view of the mappings contained in this map. |
| 13 | boolean equals(Object o) | Compares the specified object with this map. |
| 14 | int hashCode() | Returns hashcode value of this map. |

# **How HashMap Works Internally In Java?**

### **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.

**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

}

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

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

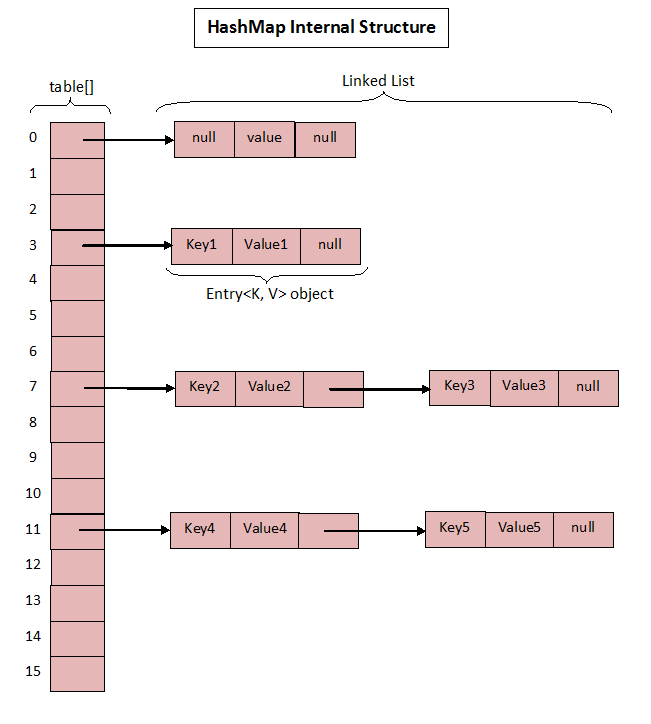
**next** : It holds the pointer to next key-value pair. **This attribute makes the key-value pairs stored as a linked list.**

**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.

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

### 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.

Note: how put method works:

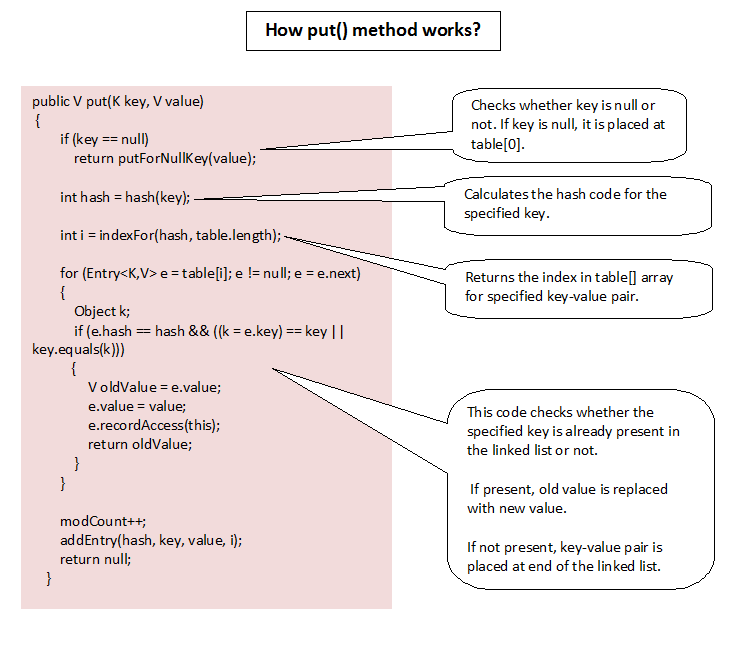
Step 1 : First checks whether the key is null or not. If the key is null, it calls putForNullKey() method. table[0] is always reserved for null key. Because, hash code of null is 0.

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.



## Performance of HashMap

The performance of HashMap depends on 2 parameters which are named as follows:

1. Initial Capacity
2. Load Factor

***1. Initial Capacity****– It is the capacity of HashMap at the time of its creation (It is the number of buckets a HashMap can hold when the HashMap is instantiated). In java, it is 2^4=16 initially, meaning it can hold 16 key-value pairs.*

***2. Load Factor****– It is the percent value of the capacity after which the capacity of Hashmap is to be increased (It is the percentage fill of buckets after which Rehashing takes place). In java, it is 0.75f by default, meaning the rehashing takes place after filling 75% of the capacity.*

***3. Threshold****– It is the product of Load Factor and Initial Capacity. In java, by default, it is (16 \* 0.75 = 12). That is, Rehashing takes place after inserting 12 key-value pairs into the HashMap.*

***4. Rehashing****– It is the process of doubling the capacity of the HashMap after it reaches its Threshold. In java, HashMap continues to rehash(by default) in the following sequence – 2^4, 2^5, 2^6, 2^7, …. so on.*

If the initial capacity is kept higher then rehashing will never be done. But by keeping it higher increases the time complexity of iteration. So it should be chosen very cleverly to increase performance. The expected number of values should be taken into account to set the initial capacity. The most generally preferred load factor value is 0.75 which provides a good deal between time and space costs. The load factor’s value varies between 0 and 1.

4 different ways:

**import** java.util.HashMap;

**public** **class** ExampleOne

{

**public** **static** **void** main(String[] args)

    {

        //1. Creating HashMap with default initial capacity and load factor

        HashMap<String, Integer> map1 = **new** HashMap<String, Integer>();

        //2. Creating HashMap with 30 as initial capacity

        HashMap<String, Integer> map2 = **new** HashMap<String, Integer>(30);

        //3. Creating HashMap with 30 as initial capacity and 0.5 as load factor

        HashMap<String, Integer> map3 = **new** HashMap<String, Integer>(30, 0.5f);

        //4. Creating HashMap by copying another HashMap

        HashMap<String, Integer> map4 = **new** HashMap<String, Integer>(map1);

    }

}