HashMap<K, V>

* HashMap<K, V> is hash table based implementation. HashMap<K, V> is concrete implementation of Map<K, V> interface. Read about Map<K, V> interface [here](http://data-structure-learning.blogspot.com/2015/05/java-collections-part-3map-interface.html) and [here](http://data-structure-learning.blogspot.com/2015/05/java-collections-part-4map-interface.html).
* This implementation permits null key and null values. Multiple null values can be used but only single null key is allowed.
* HashMap<K, V> is unsynchronized so it should not be used in multi-threading environment. But we can use thread-safe version of HashMap<K, V> ConcurrentHashMap<K, V>(null keys and values are not allowed) or use Collections.synchronizedMap(new HashMap<K, V>) (iterator must be synchronized externally.).
* HashMap<K, V> has its initial capacity and load factor. Initial capacity means capacity of underlying table use when HashMap<K, V> instance was created. Load factor is a measure of how much capacity of hash table is allowed before it is rebuilt. Rebuilding of hash table is costly operation but it is necessary evil one must go through.
* HashMap<K, V> implementation is has also one very important work to do that is generate hashCode for key. It needs to make sure that the hashCode generated is dispersed across the table.

The iterators returned by 3 collection views are fail-fast. Read about [fail-fast and how to avoid it](http://data-structure-learning.blogspot.com/2015/05/fail-fast-iterator.html). Read about [Fail-Safe](http://data-structure-learning.blogspot.com/2015/05/fail-safe-iterator.html) and [ConcurrentModificationException](http://data-structure-learning.blogspot.com/2015/05/concurrentmodificationexception.html). [Difference between Fail-Fast and Fail-Safe](http://data-structure-learning.blogspot.com/2015/05/difference-between-fail-safe-iterator.html).

We will now understand method of this class one by one.

1. First is put(K key, V value) method.

put(..) method is used to insert key and value mapping in to the underlying hash table. There is always one key and one value in one entry to map. Keys are unique per map instance. Values can be repeated for different keys. If map contains the key-value mapping then old value is replaced by new value specified in put(..) method.

/\*\*

\* We use this method to demonstrate working of

\* put method of HashMap<K, V> class

\* \*/

**public** **static** Map<String, String> populateMap() {

Map<String, String> map = **new** HashMap<String, String>();

map.put("Effective Java", "Joshua Bloch");

map.put("Neo4j in Action", "Nicki");

map.put("Lucene in Action", "Micheal");

map.put("Complete Reference to Java", "Herbert");

map.put("Google Guava", "Bill");

**return** map;

}

Map prints

{Neo4j in Action=Nicki, Complete Reference to Java=Herbert, Effective Java=Joshua Bloch, Google Guava=Bill, Lucene in Action=Micheal}

Look at the output. We inserted Key “Effective Java” first but while printing it is third. Why so? Because HashMap<K, V> is unordered collection. It inserts key-value mappings into the table based in hashCode. **So it is unordered in terms of natural ordering but ordered in terms of hashCode.**

1. Second get(K key) method

get(K key) is used to search the key in table and return the corresponding value to it. But we need to check whether the key exists in table or not. For that we use containsKey(..) method to check whether key exists in table. It returns boolean value true if table has key else false if no key in table.

/\*\*

\* get method demo

\* check if key exists in map

\* if yes then return value using get method key as parameter returns value

\* else return null.

\* \*/

**public** **static** String getDemo(Map<String, String> map, String bookName) {

**if** (map.containsKey(bookName)) {

**return** map.get(bookName);

}

**return** **null**;

}

1. Third, putAll(..) method. Well this is good method and accepts a parameter map. Internally, it will uses put method for all the mappings provided by parameter map. Below is the demo for putAll(..)

/\*\*

\* We use this method to demonstrate working of

\* put method of HashMap<K, V> class

\* \*/

**public** **static** Map<String, String> populateMap() {

Map<String, String> map = **new** HashMap<String, String>();

map.put("Effective Java", "Joshua Bloch");

map.put("Neo4j in Action", "Nicki");

map.put("Lucene in Action", "Micheal");

map.put("Complete Reference to Java", "Herbert");

map.put("Google Guava", "Bill");

**return** map;

}

/\*\*

\* We use this method to demonstrate working of

\* put method of HashMap<K, V> class

\* \*/

**public** **static** Map<String, String> populateMapForDatabases() {

Map<String, String> map = **new** HashMap<String, String>();

map.put("MySQL", "SQL");

map.put("Oracle 9i", "SQL");

map.put("Neo4j", "NoSQL");

map.put("MongoDB", "NoSQL");

map.put("Oracle 10g", "MySQL");

**return** map;

}

**public** **static** Map<String, String> putAllDemo(Map<String, String> bookAuthors, Map<String, String> databases){

Map<String, String> booksAndDatabases = **new** HashMap<String, String>();

booksAndDatabases.putAll(bookAuthors);

booksAndDatabases.putAll(databases);

**return** booksAndDatabases;

}

It prints

{Neo4j in Action=Nicki, Oracle 9i=SQL, Neo4j=NoSQL, MySQL=SQL, Complete Reference to Java=Herbert, Effective Java=Joshua Bloch, Google Guava=Bill, MongoDB=NoSQL, Lucene in Action=Micheal, Oracle 10g=MySQL}

1. Fourth, clear() method.

This method is used to remove all the mappings from the map. After the call returns the map will be empty.

/\*\*

\* We use this method to demonstrate working of

\* clear() method.

\* \*/

**public** **static** **void** isEmptyDemo() {

Map<String, String> map = **new** HashMap<String, String>();

map.put("Effective Java", "Joshua Bloch");

map.put("Neo4j in Action", "Nicki");

map.put("Lucene in Action", "Micheal");

map.put("Complete Reference to Java", "Herbert");

map.put("Google Guava", "Bill");

map.clear();

}

1. Fifth, containsValue(Object value)

This method returns true if table contains value else returns false.

/\*\*

\* We use this method to demonstrate working of

\* containValue() method.

\* \*/

**public** **static** **void** containValueDemo() {

Map<String, String> map = **new** HashMap<String, String>();

map.put("Effective Java", "Joshua Bloch");

map.put("Neo4j in Action", "Nicki");

map.put("Lucene in Action", "Micheal");

map.put("Complete Reference to Java", "Herbert");

map.put("Google Guava", "Bill");

System.***out***.println(map.containsValue("Joshua Bloch"));

}

It prints true.

1. Sixth, isEmpty().

Returns true if map is empty else returns false.

**boolean** empty=map.isEmpty();

1. Seventh, remove(Object key)

Remove searches for the key in table and returns the value corresponding to this key. Check for map size after remove.

/\*\*

\* We use this method to demonstrate working of

\* remove() method.

\* \*/

**public** **static** **void** removeDemo() {

Map<String, String> map = **new** HashMap<String, String>();

map.put("Effective Java", "Joshua Bloch");

map.put("Neo4j in Action", "Nicki");

map.put("Lucene in Action", "Micheal");

map.put("Complete Reference to Java", "Herbert");

map.put("Google Guava", "Bill");

System.***out***.println(map.size());//prints 5

map.remove("Google Guava");

System.***out***.println(map.size());//prints 4

}

1. Eighth, size()

Returns the int value, which is size of the map which has key-value mappings.

**int** size=map.size();

Now there are 3 Collection view for Map. I have written a post on [5 ways to iterate on Map](http://data-structure-learning.blogspot.com/2015/05/different-ways-to-traverse-map.html). Refer that for these 3 methods.