**10 things a java developer should know about HashMap**

* Published on May 25, 2015

**[Chinmay Parekh](https://www.linkedin.com/in/chinmay-parekh-442bb38/)**

<https://www.linkedin.com/pulse/10-things-java-developer-should-know-hashmap-chinmay-parekh/>

This writeup assumes that you have a basic familiarity with Java’s HashMap and dives into some of its lesser known but very important characteristics.Let me start with a very brief primer before we take a deep dive.

**Introduction:**

Basic syntax:

HashMap<String, Object> hashMap = new HashMap<>();

hashMap.put(“key”, “value");

Philosophy behind hashMap:

If you are looking for an element in a "list", the order of lookup would be proportional to the length of the list,

however, if you split this list into multiple mini-lists AND IF you can quickly tell in which mini-list this element may reside, then the time-complexity for look-up can be reduced greatly. That is what hashMap does, it builds an array of these mini-lists.

So with basics out of the way, here are 10 things every java programmer should know about HashMap.

**(1) Internals of lookup process:**

Lookup process is at the heart of HashMap and almost all the complexity of hashMap lies here. The lookup process consists of 2 steps:

Step# 1: Quickly determine the bucket number in which this element may reside (using key.hashCode()).

Step# 2: Go over the mini-list and return the element that matches the key (using key.equals()).

Note the the value object is not used in any of the calculations within hashMap.

**(2) Immutability of keys:**

A class that we plan to use as a “key” in hashMap needs to follow certain restrictions.

If the object which is used as key in hashMap is modified, then we may have problem retrieving values from hashMap.

Let's say I use an object as key, and put this key and associated value into hashMap. Later I modify one of the properties of this key. If hashCode() and equals() method make use of this property, then we may not find this key in hashMap. If the property being modified is not being used by hashCode() and equals(), then we will still be able to find the key in hashMap.

To do away with this issue altogether, recommendation is to use immutable classes as keys.

Note, that even if only one of the methods hashCode() or equals() return different results(when a property is modified), the key becomes useless from the lookup perspective.

**(3) Load factor and resize:**

When a hashMap resizes, it will double in size and create a new instance and populate it.

When new hashHap is being populated, the linkedList associated with each bucket of source hashMap is iterated and nodes are copied to the destination bucket. However, note that these new nodes are prepended to the head of the destination linkedList. So resizing has an side effect of reversing the order of the items in the list. Default load factor for hashMap is 0.75.

**(4) Worst-case performance:**

In the worst case, a hashMap reduces to a linkedList.

However with Java 8, there is a change,

Java 8 intelligently determines if we are running in the worst-case performance scenario and converts the list into a binary search tree.

**(5) Collisions:**

Collisions happen when 2 distinct keys generate the same hashCode() value. Multiple collisions are the result of bad hashCode() algorithm.

The more the collisions the worse the performance of the hashMap.

There are many collision-resolution strategies - chaining, double-hashing, clustering.

However, java has chosen chaining strategy for hashMap, so in case of collisions, items are chained together just like in a linkedList.

**(6) Adding duplicate entries into hashMap:**

Attempt to put the same key with a different value, will overwrite the old value.

1: hashHap.put(myKey, oldValue);

2: hashHap.put(myKey, newValue);

3: hashMap.get(myKey);*// This line will return newValue*

Line# 2, above will essentially over-write oldValue with newValue. so hashMap.put() is actually "add or overwrite".

**(7) Concurrency:**

Do not use HashMap in multi-threaded environment.

What if 2 threads starts resizing the hashMap at the same time?

**(8) Map of maps:**

HashMap of hashMaps are very popular.

Pseudo code below to give an idea:

Map<String, Map<String, Object>> multiHashMap = new HashMap<>();

Map<String, Object> myHashMap1;

Map<String, Object> myHashMap2;

multiHashMap.put(“one”, myHashMap1);

multiHashMap.put(“two”, myHashMap2);

Also, "multimap" is a common dataStructure, where value is a collection.

You use the key to retrieve the collection and the manipulate the collection.

Map<Double,List<Object>> multiMap = new TreeMap<Double,List<Object>>();

**(9) Some specialized hashMaps for specific purposes:**

- ConcurrentHashMap: HashMap to be used in multithreaded applications.

- EnumMap: HashMap with Enum values as keys.

- LinkedHashMap: HashMap with predictable iteration order (great for FIFO/LIFO caches)

**(10) Code samples:**

It is really insightful to jump into the source, here is how hashCode() and equals() are implemented in jdk.

(a) Here is how **[String.hashCode()](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/String.java" \l "String.hashCode%28%29" \t "_blank) and [String.equals()](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/String.java" \l "String.equals%28java.lang.Object%29" \t "_blank)** are implemented.

(b) Here is how **[Object.hashCode()](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/Object.java" \l "Object.hashCode%28%29" \t "_blank) and [Object.equals()](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/Object.java" \l "Object.equals%28java.lang.Object%29" \t "_blank)**are implemented.