**Rule:**

Two equal objects should have same hashcodes, vice-versa is not mandatory.

If you don’t follow this rule and if two equal objects have diff hashcodes, then these two objects might get stored at different places in an array of buckets. When you say map.contains(…), it may return false. It will search the object by its hash code. It will find an object at one of the two locations, but found location may not have the same object (using equals).

So, if you have overridden equals method, you should override hashCode method as well.

<https://examples.javacodegeeks.com/core-java/util/hashmap/hashmap-changes-in-java-8/>

<http://winterbe.com/posts/2015/05/22/java8-concurrency-tutorial-atomic-concurrent-map-examples/>

The way java.util.HashMap entries are indexed and stored has changed in the Java 8 update.  
Hash elements use balanced trees instead of linked lists under certain circumstances now.  
<http://coding-geek.com/how-does-a-hashmap-work-in-java/>

**Till Java7**, HashMap uses **‘Separate Chaining’** mechanism to store elements of HashMap. In this mechanism, HashMap used to keep an array of buckets. Each bucket used to have elements stored as linked list. LinkedList takes O(n) for access in worst case. If for some reason, all the elements are stored in the same bucket, then you have to go through entire linked list of that bucket, if you need to find the last element stored in that linked list.

In Java8, code has become more complex. Number of lines in the code has been doubled. So, what's good about Java8's HashMap?  
It keeps **Balanced Search Trees(BSTs)** in the buckets instead of LinkedLists, if number of elements to be stored in a bucket is more than certain number. Search in BST takes **O(log n)**, which is a lot less than LinkedList.  
  
- If for a given index (bucket) in the inner table there are more than 8 nodes, the linked list is transformed into a **red black tree**

- If for a given index (bucket) in the inner table there are less than 6 nodes, the tree is transformed into a linked list  
  
The fix has been implemented in the classes  
 java.util.HashMap,  
 java.util.LinkedHashMap and  
 java.util.concurrent.ConcurrentHashMap  
They haven't considered WeakHashMap for this optimization.  
  
newly added methods

forEach(consumer)

search

reduce  
 replace(key, value)  
 replaceAll(key, function)  
 compute(key, BiFunction)  
 computeIfAbsent(key, Function)  
 computeIfPresent(key, BiFunction)  
 merge

putIfAbsent

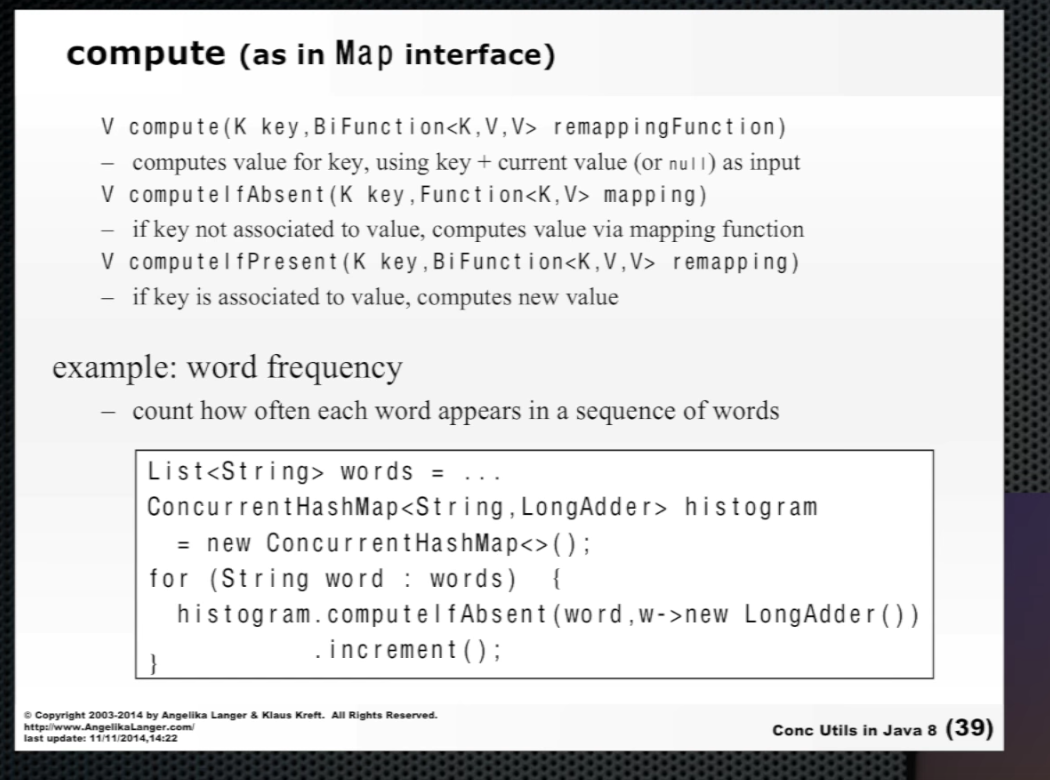
compute methods take a function as parameter that takes key and value and returns a computed value.

**Compute methods are just like replace method with slight variations.**

|  |  |
| --- | --- |
| map.replace("1", "one-new") | If key="1" exists with null/non-null value, then only it replaces its value. It doesn't create a new key "1", if it doesn't exist |
| map.replace("1", "one", "one-new") | Replaces the value of a key "1" to "one-new", if current value matches to "one" |
| map.compute("1", (k, v) -> "value of "+k) | It creates a key, if it doesn’t exist, otherwise it replaces computed value. If computed value is null, then it removes the key, if it exists already. |
| map.computeIfAbsent("4", (k) -> k + "-new") | if key doesn't exist or value of the key is null, then insert a key with computed value, if computed value is not null. |
| map.computeIfPresent("3", (k, v) -> v + "-new computed value") | if key exist and its value is not null, then replace that value with new computed value. If computed value is null, then it removes the key. |

If you see carefully, replace method doesn’t care whether existing value of a key is null or non-null, but compute methods does care about it. Compute methods will remove the key, if new computed value is null.

<https://www.youtube.com/watch?v=Q_0_1mKTlnY>



computeIfAbsent is a special method that is related to Memoization.

Integer doubleValue(Integer x) {  
 return x \* 2;  
}  
  
Memoized form of above method:

Normally, you use Map to memorize the result.  
  
Map<Integer, Integer> cache = new ConcurrentHashMap<>();  
Integer doubleValue(Integer x) {  
 if (cache.containsKey(x)) {  
 return cache.get(x);  
 } else {  
 Integer result = x \* 2;  
 cache.put(x, result) ;  
 return result;  
 }  
}  
  
Using Java 8, you can even make it simpler.  
  
Map<Integer, Integer> cache = new ConcurrentHashMap<>();  
Integer doubleValue(Integer x) {  
 return cache.computeIfAbsent(x, x1 -> x1 \* 2);  
}

**merge**

<http://www.buggybread.com/2014/10/java-8-map-merge-method.html>

Merge the specified value to the existing Value using the Specified function for the Specified Key.

merge([Key],[New Value],[Function([Current Value],[New Value]]);

Map<String,String> strMap = new HashMap<String,String>();  
strMap.put("Key1","Value1");  
strMap.put("Key2", "Value2");

String str = strMap.merge("Key1","Value56",(v1,v2)->v1.substring(3).concat(v2));

System.out.println(str); // prints ue1Value56  
System.out.println(strMap); // prints {Key2=Value2, Key1=ue1Value56}

