The Java HashMap Under the Hood

Last modified: May 5, 2018

by baeldung (http://www.baeldung.com/author/baeldung/)

Java (http://www.baeldung.com/category/java/)

Java Collections (http://www.baeldung.com/tag/collections/)

1. Overview

In this article, we are going to explore the most popular implementation of *Map* interface from the Java Collections Framework.

Before we get started with the implementation, it's important to point out that the primary *List* and *Set* collection interfaces extend *Collection* but *Map* does not.

Simply put, the *HashMap* stores values by key and provides APIs for adding, retrieving and manipulating stored data in various ways. The implementation is **based on the the principles of a hashtable**, which sounds a little complex at first but is actually very easy to understand.

Key-value pairs are stored in what is known as buckets which together make up what is called a table, which is actually an internal array.

Once we know the key under which an object is stored or is to be stored, **storage and retrieval operations occur in constant time**, *O(1)* in a well-dimensioned hash map.

To understand how hash maps work under the hood, one needs to understand the storage and retrieval mechanism employed by the *HashMap.* We'll focus a lot on these.

Finally, ***HashMap* related questions are quite common in interviews**, so this is a solid way to either prepare an interview or prepare for it.

2. The *put()* API

To store a value in a hash map, we call the *put* API which takes two parameters; a key and the corresponding value:

```
1 V put(K key, V value);
```

When a value is added to the map under a key, the *hashCode()* API of the key object is called to retrieve what is known as the initial hash value.

To see this in action, let us create an object that will act as a key. We will only create a single attribute to use as a hash code to simulate the first phase of hashing:

```
public class MyKey {
    private int id;

    @Override
    public int hashCode() {
        System.out.println("Calling hashCode()");
        return id;
    }

    // constructor, setters and getters
}
```

We can now use this object to map a value in the hash map:

```
@Test
public void whenHashCodeIsCalledOnPut_thenCorrect() {
    MyKey key = new MyKey(1);
    Map<MyKey, String> map = new HashMap<>();
    map.put(key, "val");
}
```

Nothing much happening in the above code, but pay attention to the console output. Indeed the *hashCode* method gets invoked:

```
Calling hashCode()
```

Next, the *hash()* API of the hash map is called internally to compute the final hash value using the initial hash value.

This final hash value ultimately boils down to an index in the internal array or what we call a bucket location.

The *hash* function of *HashMap* looks like this:

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

What we should note here is only the use of the hash code from the key object to compute a final hash value.

While inside the *put* function, the final hash value is used like this:

```
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}
```

Notice that an internal *putVal* function is called and given the final hash value as the first parameter.

One may wonder why the key is again used inside this function since we have already used it to compute the hash value.

The reason is that **hash maps store both key and value in the bucket location as a *Map.Entry* object**.

As discussed before, all Java collections framework interfaces extend *Collection* interface but *Map* does not. Compare the declaration of Map interface we saw earlier to that of *Set* interface:

```
public interface Set<E> extends Collection<E>
```

The reason is that **maps do not exactly store single elements as do other collections but rather a collection of key-value pairs.**

So the generic methods of *Collection* interface such as *add*, *toArray* do not make sense when it comes to *Map*.

The concept we have covered in the last three paragraphs makes for one of the **most popular Java Collections Framework interview questions**. So, it's worth understanding.

One special attribute with the hash map is that it accepts *null* values and null keys:

```
@Test
public void givenNullKeyAndVal_whenAccepts_thenCorrect(){
    Map<String, String> map = new HashMap<>();
    map.put(null, null);
}
```

When a null key is encountered during a *put* operation, it is automatically assigned a final hash value of *0*, which means it becomes the first element of the underlying array.

This also means that when the key is null, there is no hashing operation and therefore, the *hashCode* API of the key is not invoked, ultimately avoiding a null pointer exception.

During a *put* operation, when we use a key that was already used previously to store a value, it returns the previous value associated with the key:

```
@Test
public void givenExistingKey_whenPutReturnsPrevValue_thenCorrect() {
    Map<String, String> map = new HashMap<>();
    map.put("key1", "val1");

    String rtnVal = map.put("key1", "val2");

    assertEquals("val1", rtnVal);
}
```

otherwise, it returns *null:*

```
@Test
public void givenNewKey_whenPutReturnsNull_thenCorrect() {
    Map<String, String> map = new HashMap<>();

    String rtnVal = map.put("key1", "val1");

    assertNull(rtnVal);
}
```

When *put* returns null, it could also mean that the previous value associated with the key is null, not necessarily that it's a new key-value mapping:

```
@Test
public void givenNullVal_whenPutReturnsNull_thenCorrect() {
    Map<String, String> map = new HashMap<>();

    String rtnVal = map.put("key1", null);

    assertNull(rtnVal);
}
```

The *containsKey* API can be used to distinguish between such scenarios as we will see in the next subsection.

3. The *get* API

To retrieve an object already stored in the hash map, we must know the key under which it was stored. We call the *get* API and pass to it the key object:

```
@Test
public void whenGetWorks_thenCorrect() {
    Map<String, String> map = new HashMap<>();
    map.put("key", "val");

    String val = map.get("key");

    assertEquals("val", val);
}
```

Internally, the same hashing principle is used. *The hashCode()* API of the key object is called to obtain the initial hash value:

```
@Test
public void whenHashCodeIsCalledOnGet_thenCorrect() {
    MyKey key = new MyKey(1);
    Map<MyKey, String> map = new HashMap<>();
    map.put(key, "val");
    map.get(key);
}
```

This time, the *hashCode* API of *MyKey* is called twice; once for *put* and once for *get*:

```
Calling hashCode()
Calling hashCode()
```

This value is then rehashed by calling the internal *hash()* API to obtain the final hash value.

As we saw in the previous section, this final hash value ultimately boils down to a bucket location or an index of the internal array.

The value object stored in that location is then retrieved and returned to the calling function.

When the returned value is null, it could mean that the key object is not associated with any value in the hash map:

```
@Test
public void givenUnmappedKey_whenGetReturnsNull_thenCorrect() {
    Map<String, String> map = new HashMap<>();

    String rtnVal = map.get("key1");

    assertNull(rtnVal);
}
```

Or it could simply mean that the key was explicitly mapped to a null instance:

```java
@Test
public void givenNullVal_whenRetrieves_thenCorrect() {
    Map<String, String> map = new HashMap<>();
    map.put("key", null);

    String val=map.get("key");

    assertNull(val);
}
```

To distinguish between the two scenarios, we can use the *containsKey* API, to which we pass the key and it returns true if and only if a mapping was created for the specified key in the hash map:

```java
@Test
public void whenContainsDistinguishesNullValues_thenCorrect() {
    Map<String, String> map = new HashMap<>();

    String val1 = map.get("key");
    boolean valPresent = map.containsKey("key");

    assertNull(val1);
    assertFalse(valPresent);

    map.put("key", null);
    String val = map.get("key");
    valPresent = map.containsKey("key");

    assertNull(val);
    assertTrue(valPresent);
}
```

For both cases in the above test, the return value of the *get* API call is null but we are able to distinguish which one is which.

4. Collection Views In *HashMap*

HashMap offers three views that enable us to treat its keys and values as another collection. We can get a set of all **keys of the map**:

```
@Test
public void givenHashMap_whenRetrievesKeyset_thenCorrect() {
    Map<String, String> map = new HashMap<>();
    map.put("name", "baeldung");
    map.put("type", "blog");

    Set<String> keys = map.keySet();

    assertEquals(2, keys.size());
    assertTrue(keys.contains("name"));
    assertTrue(keys.contains("type"));
}
```

The set is backed by the map itself. So **any change made to the set is reflected in the map**:

```
@Test
public void givenKeySet_whenChangeReflectsInMap_thenCorrect() {
    Map<String, String> map = new HashMap<>();
    map.put("name", "baeldung");
    map.put("type", "blog");

    assertEquals(2, map.size());

    Set<String> keys = map.keySet();
    keys.remove("name");

    assertEquals(1, map.size());
}
```

We can also obtain a **collection view of the values**:

```
@Test
public void givenHashMap_whenRetrievesValues_thenCorrect() {
    Map<String, String> map = new HashMap<>();
    map.put("name", "baeldung");
    map.put("type", "blog");

    Collection<String> values = map.values();

    assertEquals(2, values.size());
    assertTrue(values.contains("baeldung"));
    assertTrue(values.contains("blog"));
}
```

Just like the key set, any **changes made in this collection will be reflected in the underlying map**.

Finally, we can obtain a **set view of all entries** in the map:

```
@Test
public void givenHashMap_whenRetrievesEntries_thenCorrect() {
    Map<String, String> map = new HashMap<>();
    map.put("name", "baeldung");
    map.put("type", "blog");

    Set<Entry<String, String>> entries = map.entrySet();

    assertEquals(2, entries.size());
    for (Entry<String, String> e : entries) {
        String key = e.getKey();
        String val = e.getValue();
        assertTrue(key.equals("name") || key.equals("type"));
        assertTrue(val.equals("baeldung") || val.equals("blog"));
    }
}
```

Remember that a hash map specifically contains unordered elements, therefore we assume any order when testing the keys and values of entries in the *for each* loop.

Many times, you will use the collection views in a loop as in the last example, and more specifically using their iterators.

Just remember that the **iterators for all the above views are *fail-fast***.

If any structural modification is made on the map, after the iterator has been created, a concurrent modification exception will be thrown:

```
@Test(expected = ConcurrentModificationException.class)
public void givenIterator_whenFailsFastOnModification_thenCorrect() {
    Map<String, String> map = new HashMap<>();
    map.put("name", "baeldung");
    map.put("type", "blog");

    Set<String> keys = map.keySet();
    Iterator<String> it = keys.iterator();
    map.remove("type");
    while (it.hasNext()) {
        String key = it.next();
    }
}
```

The only **allowed structural modification is a *remove*** operation performed through the iterator itself:

```
public void givenIterator_whenRemoveWorks_thenCorrect() {
    Map<String, String> map = new HashMap<>();
    map.put("name", "baeldung");
    map.put("type", "blog");

    Set<String> keys = map.keySet();
    Iterator<String> it = keys.iterator();

    while (it.hasNext()) {
        it.next();
        it.remove();
    }

    assertEquals(0, map.size());
}
```

The final thing to remember about these collection views is the performance of iterations. This is where a hash map performs quite poorly compared with its counterparts linked hash map and tree map.

Iteration over a hash map happens in worst case *O(n)* where n is the sum of its capacity and the number of entries.

5. HashMap Performance

The performance of a hash map is affected by two parameters: *Initial Capacity* and *Load Factor*. The capacity is the number of buckets or the underlying array length and the initial capacity is simply the capacity during creation.

The load factor or LF, in short, is a measure of how full the hash map should be after adding some values before it is resized.

The default initial capacity is *16* and default load factor is *0.75*. We can create a hash map with custom values for initial capacity and LF:

```
Map<String,String> hashMapWithCapacity=new HashMap<>(32);
Map<String,String> hashMapWithCapacityAndLF=new HashMap<>(32, 0.5f);
```

The default values set by the Java team are well optimized for most cases. However, if you need to use your own values, which is very okay, you need to understand the performance implications so that you know what you are doing.

When the number of hash map entries exceeds the product of LF and capacity, then **rehashing** occurs i.e. **another internal array is created with twice the size of the initial one and all entries are moved over to new bucket locations in the new array**.

A **low initial capacity** reduces space cost but **increases the frequency of rehashing**. Rehashing is obviously a very expensive process. So as a rule, if you anticipate many entries, you should set a considerably high initial capacity.

On the flip side, if you set the initial capacity too high, you will pay the cost in iteration time. As we saw in the previous section.

So **a high initial capacity is good for a large number of entries coupled with little to no iteration**.

A **low initial capacity is good for few entries with a lot of iteration**.

6. Collisions in the *HashMap*

A collision, or more specifically, a hash code collision in a *HashMap*, is a situation where **two or more key objects produce the same final hash value** and hence point to the same bucket location or array index.

This scenario can occur because according to the *equals* and *hashCode* contract, **two unequal objects in Java can have the same hash code**.

It can also happen because of the finite size of the underlying array, that is, before resizing. The smaller this array, the higher the chances of collision.

That said, it's worth mentioning that Java implements a hash code collision resolution technique which we will see using an example.

Keep in mind that it's the hash value of the key that determines the bucket the object will be stored in. And so, if the hash codes of any two keys collide, their entries will still be stored in the same bucket.

And by default, the implementation uses a linked list as the bucket implementation.

The initially constant time *O(1) put* and *get* operations will occur in linear time *O(n)* in the case of a collision. This is because after finding the bucket location with the final hash value, each of the keys at this location will be compared with the provided key object using the *equals* API.

To simulate this collision resolution technique, let's modify our earlier key object a little:

```
public class MyKey {
    private String name;
    private int id;

    public MyKey(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // standard getters and setters

    @Override
    public int hashCode() {
        System.out.println("Calling hashCode()");
        return id;
    }

    // toString override for pretty logging

    @Override
    public boolean equals(Object obj) {
        System.out.println("Calling equals() for key: " + obj);
        // generated implementation
    }

}
```

Notice how we're simply returning the *id* attribute as the hash code – and thus force a collision to occur.

Also, note that we've added log statements in our *equals* and *hashCode* implementations – so that we know exactly when the logic is called.

Let's now go ahead to store and retrieve some objects that collide at some point:

```
@Test
public void whenCallsEqualsOnCollision_thenCorrect() {
    HashMap<MyKey, String> map = new HashMap<>();
    MyKey k1 = new MyKey(1, "firstKey");
    MyKey k2 = new MyKey(2, "secondKey");
    MyKey k3 = new MyKey(2, "thirdKey");

    System.out.println("storing value for k1");
    map.put(k1, "firstValue");
    System.out.println("storing value for k2");
    map.put(k2, "secondValue");
    System.out.println("storing value for k3");
    map.put(k3, "thirdValue");

    System.out.println("retrieving value for k1");
    String v1 = map.get(k1);
    System.out.println("retrieving value for k2");
    String v2 = map.get(k2);
    System.out.println("retrieving value for k3");
    String v3 = map.get(k3);

    assertEquals("firstValue", v1);
    assertEquals("secondValue", v2);
    assertEquals("thirdValue", v3);
}
```

In the above test, we create three different keys – one has a unique *id* and the other two have the same *id*. Since we use *id* as the initial hash value, **there will definitely be a collision** during both storage and retrieval of data with these keys.

In addition to that, thanks to the collision resolution technique we saw earlier, we expect each of our stored values to be retrieved correctly, hence the assertions in the last three lines.

When we run the test, it should pass, indicating that collisions were resolved and we will use the logging produced to confirm that the collisions indeed occurred:

```
storing value for k1
Calling hashCode()
storing value for k2
Calling hashCode()
storing value for k3
Calling hashCode()
Calling equals() for key: MyKey [name=secondKey, id=2]
retrieving value for k1
Calling hashCode()
retrieving value for k2
Calling hashCode()
retrieving value for k3
Calling hashCode()
Calling equals() for key: MyKey [name=secondKey, id=2]
```

Notice that during storage operations, *k1* and *k2* were successfully mapped to their values using only the hash code.

However, storage of *k3* was not so simple, the system detected that its bucket location already contained a mapping for *k2*. Therefore, *equals* comparison was used to distinguish them and a linked list was created to contain both mappings.

Any other subsequent mapping whose key hashes to the same bucket location will follow the same route and end up replacing one of the nodes in the linked list or be added to the head of the list if *equals* comparison returns false for all existing nodes.

Likewise, during retrieval, *k3* and *k2* were *equals*-compared to identify the correct key whose value should be retrieved.

On a final note, from Java 8, the linked lists are dynamically replaced with balanced binary search trees in collision resolution after the number of collisions in a given bucket location exceed a certain threshold.

This change offers a performance boost, since, in the case of a collision, storage and retrieval happen in *O(log n).*

This section is **very common in technical interviews,** especially after the basic storage and retrieval questions.

7. Conclusion

In this article, we have explored *HashMap* implementation of Java *Map* interface.

The full source code for all the examples used in this article can be found in the GitHub project (https://github.com/eugenp/tutorials/tree/master/core-java-collections).

(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-post-footer-main-1.2.0.jpg)

(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-post-footer-icn-1.0.0.png)

Learning to "Build your API **with Spring**"?

Enter your Email Address

>> Get the eBook

CATEGORIES

SPRING (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTP://WWW.BAELDUNG.COM/JACKSON)
HTTPCLIENT 4 TUTORIAL (HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/)
SPRING PERSISTENCE TUTORIAL (HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/)
SECURITY WITH SPRING (HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
THE COURSES (HTTP://COURSES.BAELDUNG.COM)
CONSULTING WORK (HTTP://WWW.BAELDUNG.COM/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
WRITE FOR BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
CONTACT (HTTP://WWW.BAELDUNG.COM/CONTACT)
COMPANY INFO (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
TERMS OF SERVICE (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)

PRIVACY POLICY (HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY)

EDITORS (HTTP://WWW.BAELDUNG.COM/EDITORS)

MEDIA KIT (PDF) (HTTPS://S3.AMAZONAWS.COM/BAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF)