**1. Overview**

**In this article, we'll see how to use *HashMap* in Java, and we'll look at how it works internally.**

A class very similar to *HashMap* is *Hashtable*. Please refer to a couple of our other articles to learn more about the [*java.util.Hashtable* class](https://www.baeldung.com/java-hash-table) itself and the [differences between *HashMap* and *Hashtable*](https://www.baeldung.com/hashmap-hashtable-differences).

**2. Basic Usage**

**Let's first look at what it means that *HashMap* is a map. A map is a key-value mapping, which means that every key is mapped to exactly one value and that we can use the key to retrieve the corresponding value from a map.**

One might ask why not simply add the value to a list. Why do we need a *HashMap*? The simple reason is performance. If we want to find a specific element in a list, the time complexity is *O(n)* and if the list is sorted, it will be *O(log n)* using, for example, a binary search.

The advantage of a *HashMap* is that the time complexity to insert and retrieve a value is *O(1)* on average. We'll look at how that can be achieved later. Let's first look at how to use *HashMap*.

**2.1. Setup**

Let's create a simple class that we'll use throughout the article:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class Product {        private String name;      private String description;      private List<String> tags;        // standard getters/setters/constructors        public Product addTagsOfOtherProdcut(Product product) {          this.tags.addAll(product.getTags());          return this;      }  } |

**2.2. Put**

We can now create a *HashMap* with the key of type *String* and elements of type *Product*:

|  |  |
| --- | --- |
| 1 | Map<String, Product> productsByName = new HashMap<>(); |

And add products to our *HashMap*:

|  |  |
| --- | --- |
| 1  2  3  4 | Product eBike = new Product("E-Bike", "A bike with a battery");  Product roadBike = new Product("Road bike", "A bike for competition");  productsByName.put(eBike.getName(), eBike);  productsByName.put(roadBike.getName(), roadBike); |

**2.3. Get**

We can retrieve a value from the map by its key:

|  |  |
| --- | --- |
| 1  2 | Product nextPurchase = productsByName.get("E-Bike");  assertEquals("A bike with a battery", nextPurchase.getDescription()); |

If we try to find a value for a key that doesn't exist in the map, we'll get a *null* value:

|  |  |
| --- | --- |
| 1  2 | Product nextPurchase = productsByName.get("Car");  assertNull(nextPurchase); |

And if we insert a second value with the same key, we'll only get the last inserted value for that key:

|  |  |
| --- | --- |
| 1  2  3 | Product newEBike = new Product("E-Bike", "A bike with a better battery");  productsByName.put(newEBike.getName(), newEBike);  assertEquals("A bike with a better battery", productsByName.get("E-Bike")); |

**2.4. Null as the Key**

*HashMap* also allows us to have *null* as a key:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Product defaultProduct = new Product("Chocolate", "At least buy chocolate");  productsByName.put(null, defaultProduct);    Product nextPurchase = productsByName.get(null);  assertEquals("At least buy chocolate", nextPurchase.getDescription()); |

**2.5. Values with the Same Key**

Furthermore, we can insert the same object twice with a different key:

|  |  |
| --- | --- |
| 1  2 | productsByName.put(defaultProduct.getName(), defaultProduct);  assertSame(productsByName.get(null), productsByName.get("Chocolate")); |

**2.6. Remove a Value**

We can remove a key-value mapping from the *HashMap*:

|  |  |
| --- | --- |
| 1  2 | productsByName.remove("E-Bike");  assertNull(productsByName.get("E-Bike")); |

**2.7. Check If a Key or Value Exists in the Map**

To check if a key is present in the map, we can use the *containsKey()* method:

|  |  |
| --- | --- |
| 1 | productsByName.containsKey("E-Bike"); |

Or, to check if a value is present in the map, we can use the *containsValue()* method:

|  |  |
| --- | --- |
| 1 | productsByName.containsValue(eBike); |

Both method calls will return *true* in our example. Though they look very similar, there is an important difference in performance between these two method calls.**The complexity to check if a key exists is *O(1)*, while the complexity to check for an element is *O(n),* as it's necessary to loop over all the elements in the map.**

**2.8. Iterating over a *HashMap***

There are three basic ways to iterate over all key-value pairs in a *HashMap*.

We can iterate over the set of all keys:

|  |  |
| --- | --- |
| 1  2  3 | for(String key : productsByName.keySet()) {      Product product = productsByName.get(key);  } |

Or we can iterate over the set of all entries:

|  |  |
| --- | --- |
| 1  2  3  4  5 | for(Map.Entry<String, Product> entry : productsByName.entrySet()) {      Product product =  entry.getValue();      String key = entry.getKey();      //do something with the key and value  } |

Fiinally, we can iterate over all values:

|  |  |
| --- | --- |
| 1 | List<Product> products = new ArrayList<>(productsByName.values()); |

**3. The Key**

**We can use any class as the key in our *HashMap*. However, for the map to work properly, we need to provide an implementation for *equals()* and *hashCode().***Let's say we want to have a map with the product as the key and the price as the value:

|  |  |
| --- | --- |
| 1  2 | HashMap<Product, Integer> priceByProduct = new HashMap<>();  priceByProduct.put(eBike, 900); |

Let's implement the *equals()* and *hashCode()* methods:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | @Override  public boolean equals(Object o) {      if (this == o) {          return true;      }      if (o == null || getClass() != o.getClass()) {          return false;      }        Product product = (Product) o;      return Objects.equals(name, product.name) &&        Objects.equals(description, product.description);  }    @Override  public int hashCode() {      return Objects.hash(name, description);  } |

**Note that *hashCode()* and *equals()* need to be overridden only for classes that we want to use as map keys, not for classes that are only used as values in a map.** We'll see why this is necessary in section 5 of this article.

**4. Additional Methods as of Java 8**

Java 8 added several functional-style methods to *HashMap*. In this section, we'll look at some of these methods.

**For each method, we'll look at two examples.** The first example shows how to use the new method, and the second example shows how to achieve the same in earlier versions of Java.

As these methods are quite straightforward, we won't look at more detailed examples.

**4.1. *forEach()***

The *forEach* method is the functional-style way to iterate over all elements in the map:

|  |  |
| --- | --- |
| 1  2  3  4 | productsByName.forEach( (key, product) -> {      System.out.println("Key: " + key + " Product:" + product.getDescription());      //do something with the key and value  }); |

Prior to Java 8:

|  |  |
| --- | --- |
| 1  2  3  4  5 | for(Map.Entry<String, Product> entry : productsByName.entrySet()) {      Product product =  entry.getValue();      String key = entry.getKey();      //do something with the key and value  } |

Our article [Guide to the Java 8 *forEach*](https://www.baeldung.com/foreach-java) covers the *forEach* loop in greater detail.

**4.2. *getOrDefault()***

Using the *getOrDefault()* method, we can get a value from the map or return a default element in case there is no mapping for the given key:

|  |  |
| --- | --- |
| 1  2  3 | Product chocolate = new Product("chocolate", "something sweet");  Product defaultProduct = productsByName.getOrDefault("horse carriage", chocolate);  Product bike = productsByName.getOrDefault("E-Bike", chocolate); |

Prior to Java 8:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Product bike2 = productsByName.containsKey("E-Bike")      ? productsByName.get("E-Bike")      : chocolate;  Product defaultProduct2 = productsByName.containsKey("horse carriage")      ? productsByName.get("horse carriage")      : chocolate; |

**4.3. *putIfAbsent()***

With this method, we can add a new mapping, but only if there is not yet a mapping for the given key:

|  |  |
| --- | --- |
| 1 | productsByName.putIfAbsent("E-Bike", chocolate); |

Prior to Java 8:

|  |  |
| --- | --- |
| 1  2  3 | if(productsByName.containsKey("E-Bike")) {      productsByName.put("E-Bike", chocolate);  } |

Our article [Merging Two Maps with Java 8](https://www.baeldung.com/java-merge-maps) takes a closer look at this method.

**4.4. *merge()***

And with [*merge()*](https://www.baeldung.com/java-merge-maps)*,* we can modify the value for a given key if a mapping exists, or add a new value otherwise:

|  |  |
| --- | --- |
| 1  2  3 | Product eBike2 = new Product("E-Bike", "A bike with a battery");  eBike2.getTags().add("sport");  productsByName.merge("E-Bike", eBike2, Product::addTagsOfOtherProdcut); |

Prior to Java 8:

|  |  |
| --- | --- |
| 1  2  3  4  5 | if(productsByName.containsKey("E-Bike")) {      productsByName.get("E-Bike").addTagsOfOtherProdcut(eBike2);  } else {      productsByName.put("E-Bike", eBike2);  } |

**4.5. *compute()***

With the *compute()* method, we can compute the value for a given key:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | productsByName.compute("E-Bike", (k,v) -> {      if(v != null) {          return v.addTagsOfOtherProdcut(eBike2);      } else {          return eBike2;      }  }); |

Prior to Java 8:

|  |  |
| --- | --- |
| 1  2  3  4  5 | if(productsByName.containsKey("E-Bike")) {      productsByName.get("E-Bike").addTagsOfOtherProdcut(eBike2);  } else {      productsByName.put("E-Bike", eBike2);  } |

It's worth noting that the **methods *merge()* and *compute()* are quite similar.** The *compute() method*accepts two arguments: the *key* and a *BiFunction* for the remapping. And *merge()* accepts three parameters: the *key*, a *default value* to add to the map if the key doesn't exist yet, and a *BiFunction* for the remapping.

**5. *HashMap* Internals**

In this section, we'll look at how *HashMap* works internally and what are the benefits of using *HashMap* instead of a simple list, for example.

As we've seen, we can retrieve an element from a *HashMap* by its key. One approach would be to use a list, iterate over all elements, and return when we find an element for which the key matches. Both the time and space complexity of this approach would be *O(n)*.

**With *HashMap*, we can achieve an average time complexity of *O(1)* for the *put* and *get* operations and space complexity of *O(n)*.** Let's see how that works.

**5.1. The Hash Code and Equals**

**Instead of iterating over all its elements, *HashMap* attempts to calculate the position of a value based on its key.**

The naive approach would be to have a list that can contain as many elements as there are keys possible. As an example, let's say our key is a lower-case character. Then it's sufficient to have a list of size 26, and if we want to access the element with key ‘c', we'd know that it's the one at position 3, and we can retrieve it directly.

However, this approach would not be very effective if we have a much bigger keyspace. For example, let's say our key was an integer. In this case, the size of the list would have to be 2,147,483,647. In most cases, we would also have far fewer elements, so a big part of the allocated memory would remain unused.

***HashMap* stores elements in so-called buckets and the number of buckets is called *capacity*.** When we want to put a value in the map, *HashMap* calculates the bucket based on the key and stores the value in that bucket. To retrieve the value, *HashMap* calculates the bucket in exactly the same way.

**5.2. Collisions**

**For this to work correctly, equal keys must have the same hash, however, different keys can have the same hash**. If two different keys have the same hash, the two values belonging to them will be stored in the same bucket. Inside a bucket, values are stored in a list and retrieved by looping over all elements. The cost of this is *O(n)*.

As of Java 8 (see [JEP 180](https://openjdk.java.net/jeps/180)), the data structure in which the values inside one bucket are stored is changed from a list to a balanced tree if a bucket contains 8 or more values, and it's changed back to a list if, at some point, only 6 values are left in the bucket. This improves the performance to be *O(log n)*.

**5.3. Capacity and Load Factor**

To avoid having many buckets with multiple values, the capacity is doubled if 75% (the load factor) of the buckets become non-empty. The default value for the load factor is 75%, and the default initial capacity is 16. Both can be set in the constructor.

**5.4. Summary of *put* and *get* Operations**

Let's summarize how the *put* and *get* operations work.

**When we add an element to the map,** *HashMap* calculates the bucket. If the bucket already contains a value, the value is added to the list (or tree) belonging to that bucket. If the load factor becomes bigger than the maximum load factor of the map, the capacity is doubled.

**When we want to get a value from the map,** *HashMap* calculates the bucket and gets the value with the same key from the list (or tree).

## HashMap in Java

The HashMap class extends AbstractMap and implements Map interface. HashMap is a key and value collection in java. The HashMap stores the data in key and value format. It provides the basic implementation of the Map interface of Java. We can put a value with using a key and also we can access this value using that key. It uses a hash table to store the map. This allows the execution time of the get() and the put() methods to remain the same.

## HashMap performance Improvement Changes in Java 8

Here I am going to discuss JDK 8’s new strategy for dealing with Hash collisions. Earlier before Java 8, the performance of the HashMap was poor due to the hash collision, it degrades the performance of HashMap significantly. This change can be notifiable only when if you are using the HashMap for a large number of elements, why (see below points)?

The traversal of HashMap, get(), and other methods lookup time of HashMap have a negative impact due to hash collisions. This situation you can face when multiple keys end up in the same bucket, then values along with their keys are placed in a linked list. So, the retrieval time of elements from HashMap increases from ***O(1)*** to ***O(n)***. Because the linked list has to be traversed to get the entry in the worst case scenario.

But **Java 8** has come with the following new strategy for HashMap objects in case of high collisions.

* To address this issue, Java 8 hash elements use **balanced trees instead of linked lists** after a certain **threshold** is reached. Which means HashMap starts with storing Entry objects in a linked list but after the number of items in a hash becomes larger than a certain threshold. The hash will change from using a linked list to a balanced tree.
* Above changes ensure the performance of **O(log(n))** in worst case scenarios and O(1) with proper hashCode().
* The alternative String hash function added in Java 7 has been removed.