

Simply put, the *WeakHashMap* is a hashtable-based implementation of the *Map* interface, with keys that are of a *WeakReference* (<https://docs.oracle.com/javase/8/docs/api/java/lang/ref/WeakReference.html>) type.

An entry in a *WeakHashMap* will automatically be removed when its key is no longer in ordinary use, meaning that there is no single *Reference* (<https://docs.oracle.com/javase/8/docs/api/java/lang/ref/Reference.html>) that point to that key. When the garbage collection (GC) process discards a key, its entry is effectively removed from the map, so this class behaves somewhat differently from other *Map* implementations.

2. Strong, Soft, and Weak References

To understand how the *WeakHashMap* works, **we need to look at a *WeakReference* class** – which is the basic construct for keys in the *WeakHashMap* implementation. In Java, we have three main types of references, which we'll explain in the following sections.

2.1. Strong References

The strong reference is the most common type of *Reference* that we use in our day to day programming:

```
1 | Integer prime = 1;
```

The variable *prime* has a *strong reference* to an *Integer* object with value 1. Any object which has a strong reference pointing to it is not eligible for GC.

2.2. Soft References

Simply put, an object that has a *SoftReference* (<https://docs.oracle.com/javase/8/docs/api/java/lang/ref/SoftReference.html>) pointing to it won't be garbage collected until the JVM absolutely needs memory. Let's see how we can create a *SoftReference* in Java:

```
1 Integer prime = 1;
2 SoftReference<Integer> soft = new SoftReference<Integer>(prime);
3 prime = null;
```

The *prime* object has a strong reference pointing to it.

Next, we are wrapping *prime* strong reference into a soft reference. After making that strong reference *null*, a *prime* object is eligible for GC but will be collected only when JVM absolutely needs memory.

2.3. Weak References

The objects that are referenced only by weak references are garbage collected eagerly; the GC won't wait until it needs memory in that case.

We can create a *WeakReference* in Java in the following way:

```
1 Integer prime = 1;
2 WeakReference<Integer> soft = new WeakReference<Integer>(prime);
3 prime = null;
```

When we made a *prime* reference *null*, the *prime* object will be garbage collected in the next GC cycle, as there is no other strong reference pointing to it.

References of a *WeakReference* type are used as keys in *WeakHashMap*.

3. *WeakHashMap* as an Efficient Memory Cache

Let's say that we want to build a cache that keeps big image objects as values, and image names as keys. We want to pick a proper map implementation for solving that problem.

Using a simple *HashMap* will not be a good choice because the value objects may occupy a lot of memory. What's more, they'll never be reclaimed from the cache by a GC process, even when they are not in use in our application anymore.

Ideally, we want a *Map* implementation that allows GC to automatically delete unused objects. When a key of a big image object is not in use in our application in any place, that entry will be deleted from memory.

Fortunately, the *WeakHashMap* has exactly these characteristics. Let's test our *WeakHashMap* and see how it behaves:

```
1 WeakHashMap<UniqueImageName, BigImage> map = new WeakHashMap<>();
2 BigImage bigImage = new BigImage("image_id");
3 UniqueImageName imageName = new UniqueImageName("name_of_big_image");
4
5 map.put(imageName, bigImage);
6 assertTrue(map.containsKey(imageName));
7
8 imageName = null;
9 System.gc();
10
11 await().atMost(10, TimeUnit.SECONDS).until(map::isEmpty);
```

We're creating a *WeakHashMap* instance that will store our *BigImage* objects. We are putting a *BigImage* object as a value and an *imageName* object reference as a key. The *imageName* will be stored in a map as a *WeakReference* type.

Next, we set the *imageName* reference to be *null*, therefore there are no more references pointing to the *bigImage* object. The default behavior of a *WeakHashMap* is to reclaim an entry that has no reference to it on next GC, so this entry will be deleted from memory by the next GC process.

We are calling a *System.gc()* to force the JVM to trigger a GC process. After the GC cycle, our *WeakHashMap* will be empty:

```
1 WeakHashMap<UniqueImageName, BigImage> map = new WeakHashMap<>();
2 BigImage bigImageFirst = new BigImage("foo");
3 UniqueImageName imageNameFirst = new UniqueImageName("name_of_big_image");
4
5 BigImage bigImageSecond = new BigImage("foo_2");
6 UniqueImageName imageNameSecond = new UniqueImageName("name_of_big_image_2")
7
8 map.put(imageNameFirst, bigImageFirst);
9 map.put(imageNameSecond, bigImageSecond);
10
11 assertTrue(map.containsKey(imageNameFirst));
12 assertTrue(map.containsKey(imageNameSecond));
13
14 imageNameFirst = null;
15 System.gc();
16
17 await().atMost(10, TimeUnit.SECONDS)
18     .until(() -> map.size() == 1);
19 await().atMost(10, TimeUnit.SECONDS)
20     .until(() -> map.containsKey(imageNameSecond));
```

Note that only the *imageNameFirst* reference is set to *null*. The *imageNameSecond* reference remains unchanged. After GC is triggered, the map will contain only one entry – *imageNameSecond*.

4. Conclusion

In this article, we looked at types of references in Java to fully understand how *java.util.WeakHashMap* works. We created a simple cache that leverages behavior of a *WeakHashMap* and test if it works as we expected.

The implementation of all these examples and code snippets can be found in the GitHub project (<https://github.com/eugenp/tutorials/tree/master/java-collections-maps-2>) – which is a Maven project, so it should be easy to import and run as it is.