


# Thread-safe maps

Theory

Practice

 100% completed, 0 problems solved ▾

## Theory

🕒 15 minutes reading

Unskip this topic

Start practicing

▲  
Unskip this topic

Welcome back, traveler, on your journey to the depths of concurrency! Remember one of your friends called `HashMap`? Today it's time to know how to work with the key-value concept in a multithreaded environment. Thread-safe maps are quite popular. Imagine you implement a cache based on a map. You have two writer threads and one reader thread. If you use `HashMap`, there is no guarantee that the reader thread will see updates made by a writer thread right after these changes were made. Another problem is that some of writer thread's updates may be lost, or that the changed data will turn out in an inconsistent state.

So if your map is meant to be used by several threads, there are two paths for you: `ConcurrentHashMap` and `Collections.synchronizedMap`. Let's start with the firstborn!

### §1. Collections.synchronizedMap

`Collections.synchronizedMap` first appeared in Java 2 and it is a static method of the standard `java.util.Collections` class.

The `synchronizedMap` method inputs a classic Java map and returns a thread-safe map. The returned map is backed by the input map and contains the same elements:





```
1 Map<String, String> map = new HashMap<>();
2 map.put("a", "Apple");
3 map.put("b", "Banana");
4
5 Map<String, String> synchronizedMap =
Collections.synchronizedMap(map); // contains the same as the input map
```

Use only the returned map, otherwise, you won't be able to ensure thread safety.

Let's consider the following example. Here we have two threads: `writer` which adds non-negative numbers to the map, and the `main` thread which adds negative numbers to the map.

```
1 public static void main(String[] args) throws InterruptedException {
2     Map<Integer, String> synchronizedNumbers =
Collections.synchronizedMap(new HashMap<>());
3
4     Thread writer = new Thread(() ->
addPositiveNumbers(synchronizedNumbers));
5     writer.start();
6
7     addNegativeNumbers(synchronizedNumbers); // add negative numbers
from the main thread
8
9     writer.join(); // wait for the writer thread
1
0 }
```

### 2 required topics

-   [Collections and thread-safety.](#) ▾
-   [The Map interface](#) ▾

### Table of contents:

- 1 [Thread-safe maps](#)
- [§1. Collections.synchronizedMap](#)
- [§2. ConcurrentHashMap](#)
- [§3. Early implementation](#)
- [§4. Modern implementation](#)
- [§5. ConcurrentHashMap example](#)
- [§6. Performance comparison](#)
- [§7. Conclusion](#)
- [Discussion](#)

```

1
1      System.out.println(synchronizedNumbers.size()); // the size is
always 200_000
1
2  }
1
3
1
4  private static void addPositiveNumbers(Map<Integer, String> target)
{
1
5      for (int i = 0; i < 100_000; i++) {
1
6          target.put(i, "Number is " + i);
1
7      }
1
8  }
1
9
2
0  private static void addNegativeNumbers(Map<Integer, String> target)
{
2
1      for (int i = -100_000; i < 0; i++) {
2
2          target.put(i, "Number is " + i);
2
3      }
2
4  }

```

Everything works as intended and the resulting size of the map is always the same. However, if you experiment and use a classic non-synchronized map instead of a `synchronizedMap`, the size of the returned map can be totally unpredictable.

Here are the main features of the `Collections.synchronizedMap` to keep in mind:

- The synchronization is performed by an object.
- The methods (and as a result, all operations) of a `synchronizedMap` are protected by a lock, which provides thread-safe access.
- To iterate over a `synchronizedMap` you need to use a **synchronized block**:

```

1  synchronized (synchronizedMap) {
2      Iterator<String> iterator = synchronizedMap.keySet().iterator();
3      while (iterator.hasNext()) {
4          // do important things
5          iterator.next();
6      }
7  }

```

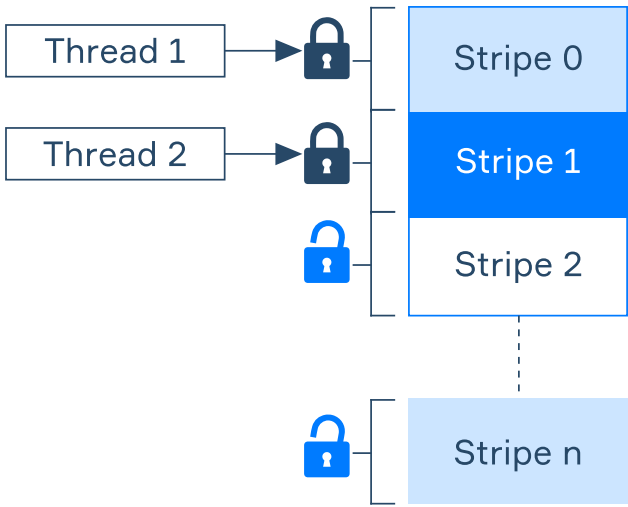
Without the synchronized block the iterator may throw `ConcurrentModificationException`. This will happen if one thread is iterating over a map and another thread is trying to modify our map.

- Only one thread at a time has access to the map, other threads are blocked.

## §2. ConcurrentHashMap

The `ConcurrentHashMap` class was created to allow multiple threads access the same map without blocking each other and, as a consequence, to increase their performance. You can find `ConcurrentHashMap` in the `java.util.concurrent` package. At this point, you may wonder: how come the `ConcurrentHashMap` class was made to be so amazing? The secret is in **Lock Striping**. This technique means that the lock mechanism occurs only on separate stripes (or buckets), but not on the whole data structure.

When a thread accesses the stripe, it locks only that stripe, leaving other stripes available. You can see what it looks like in the diagram below.



Lock Striping technique

In the history of Java, the internal realization of the `ConcurrentHashMap` class has changed quite a lot. It has been the way we know it since Java 8. Let's quickly look into what it was like until Java 7 and how it has changed later on.

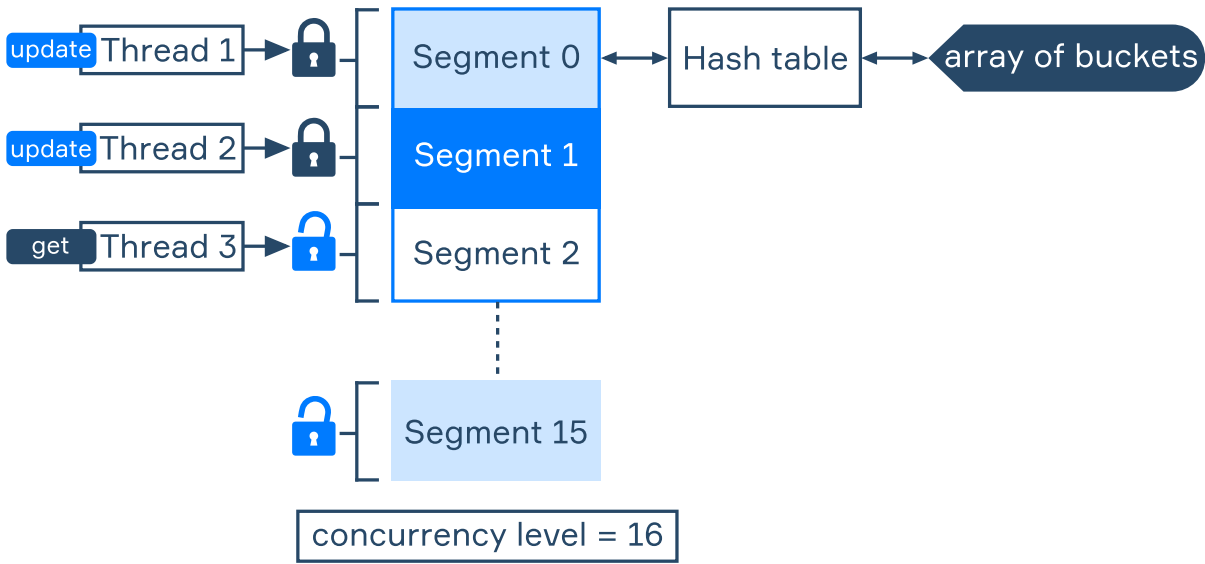
§3. Early implementation

First, there is something both of these versions have in common: **retrieval operations** (such as `get` ) do not block and give the results of the most recent update operations.

Until Java 7, `ConcurrentHashMap` was consisted of several `Segments` , where one `Segment` is a specialized and concurrently readable hash table. The number of `Segments` is given in a constructor of `ConcurrentHashMap` and called `concurrencyLevel` . By default, `concurrencyLevel` is 16.

As you can guess from the name, `concurrencyLevel` represents how many threads can update data at the same time. Also, `concurrencyLevel` is fixed, so it is recommended to choose it to accommodate the exact number of threads that will ever modify the `ConcurrentHashMap` at the same time.

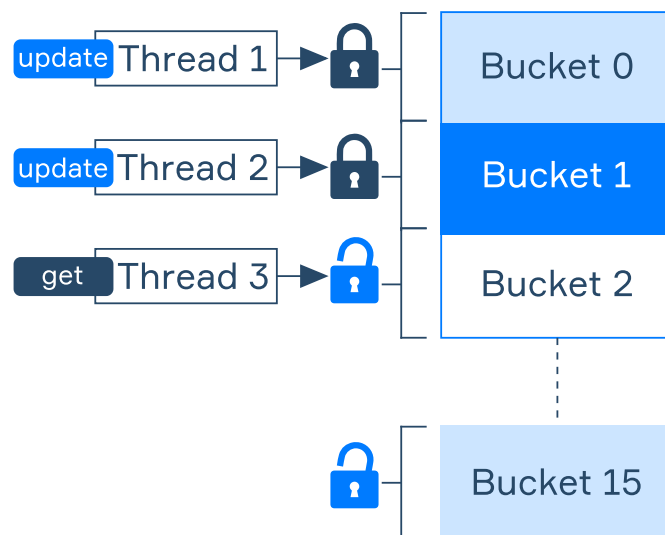
Here is a diagram to illustrate how it works:



`ConcurrencyLevel` and `Segments` are still with us but we only use them for combining with the previous versions of `ConcurrentHashMap` .

§4. Modern implementation

Since Java 8 `ConcurrentHashMap` contains 16 buckets by default, the same number as in the `HashMap` . There every bucket is a list with key-value nodes. As you can see, the structure looks more compact:



Here are the main features of `ConcurrentHashMap`:

- The synchronization is performed at the bucket level.
- Retrieval operations don't require a block.
- The iterator won't throw `ConcurrentModificationException` when one thread is iterating over a map and another thread is trying to modify it. However, there is no guarantee, that iterator will see changes made by another thread.
- The number of threads working simultaneously with the `ConcurrentHashMap` can increase if the size of the `ConcurrentHashMap` has been increased.

Note that `ConcurrentHashMap` doesn't allow using `null` as a key or a value

## §5. ConcurrentHashMap example

Here is an example of usage `ConcurrentHashMap`:

```

1  public static void main(String[] args) throws InterruptedException {
2
3      Map<Integer, String> map = new ConcurrentHashMap<>();
4
5      Thread writer1 = new Thread(() -> addPositiveNumbers(map));
6      Thread writer2 = new Thread(() -> addNegativeNumbers(map));
7
8      writer1.start();
9      writer2.start();
10
11
12      // Here, in the main thread, we can use Iterator, retrieve
values or update the map
13
14
15      writer1.join(); // wait for writer1 thread
16
17      writer2.join(); // wait for writer2 thread
18
19      System.out.println(map.size()); // the result is always the same
20
21  }
22
23  private static void addPositiveNumbers(Map<Integer, String> target)
{
24
25      for (int i = 0; i < 100_000; i++) {
26
27          target.put(i, "Number is " + i);
28
29      }
30  }

```

```
2    }
3  }
4
5  private static void addNegativeNumbers(Map<Integer, String> target)
{
2
6      for (int i = -100_000; i < 0; i++) {
2
7          target.put(i, "Number is " + i);
2
8      }
2
9  }
```

## §6. Performance comparison

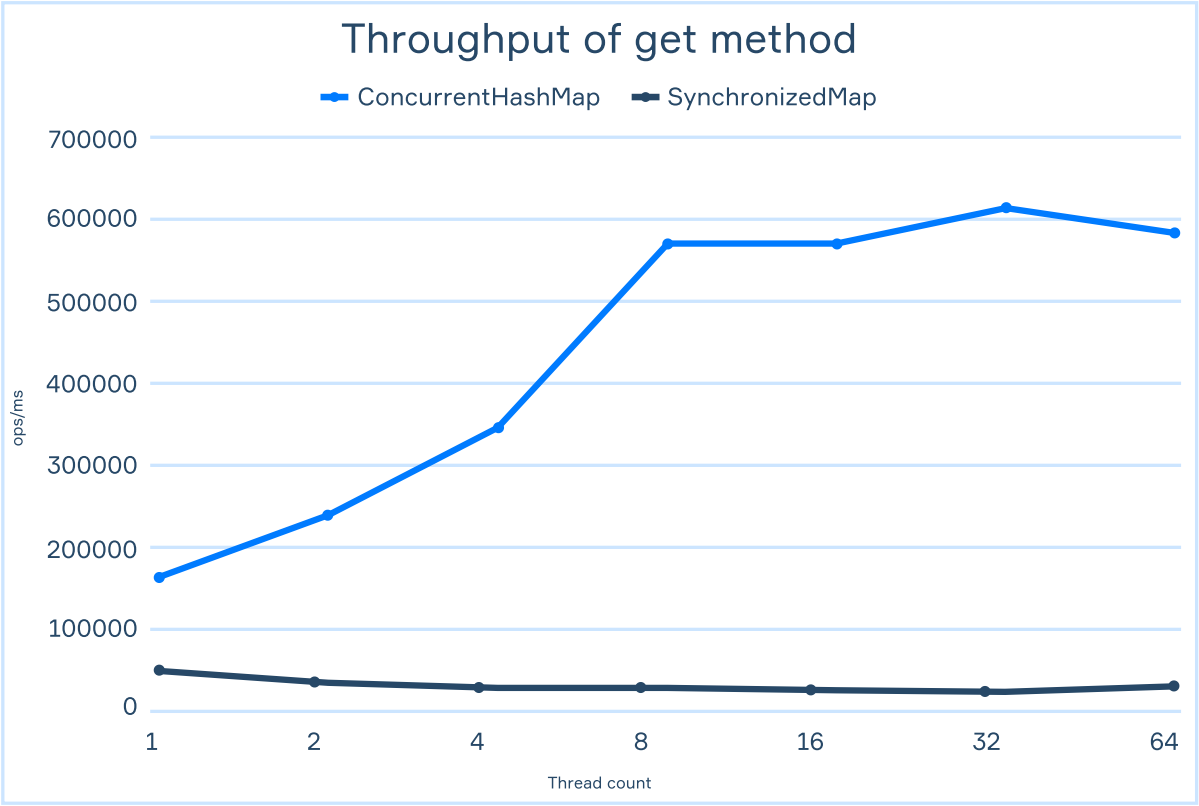
This all sounds great, but facts speak louder than words, right? Let's compare the throughput of `put` and `get` methods for `ConcurrentHashMap` and `synchronizedMap` !

To make it more interesting, we will test the throughput for JDK 11 with random keys. Both maps will have the same number of elements – 1000.

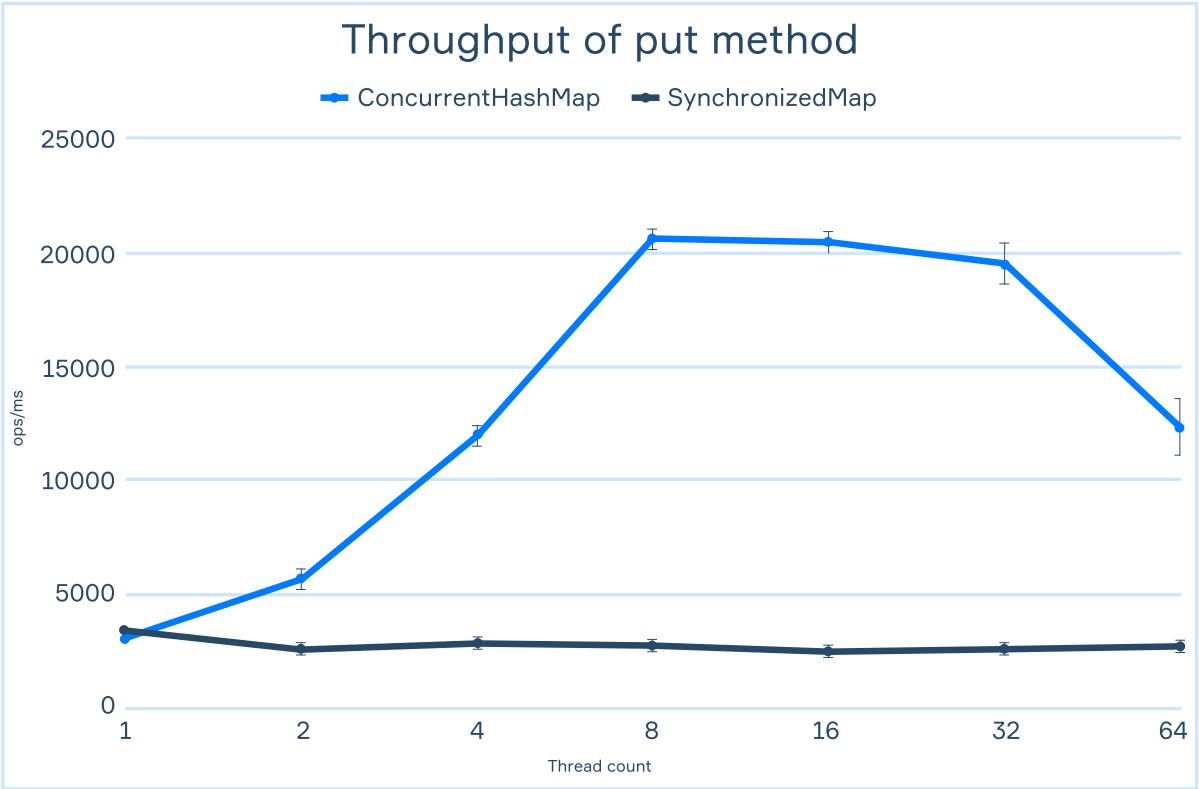
Disclaimer: the values of the charts shown are approximate. They just compare the two approaches. Do not focus on numbers, it is the trend that matters!

The representative range here is [1 - 8] threads. After that, other effects come into play.

As you see, `ConcurrentHashMap` has a much better performance in both scenarios. The non-blocking `get` operation makes `ConcurrentHashMap` faster even for a single thread. Then the throughput gradually increases, while the values of `SynchronizedMap` remain unchanged.



The `put` operation in the case of a single thread shows the same throughput, but then the situation looks similar to the `get` operation.



## §7. Conclusion

There are two thread-safe implementations of a Map interface in the standard library: `ConcurrentHashMap` and `Collections.synchronizedMap`. They use totally different ideas to achieve the same result. `Collections.synchronizedMap` inputs a classic Java map and wraps its methods by synchronized blocks. `ConcurrentHashMap` relies on a lock-stripping approach. It allows multiple threads to access different parts of the same map without blocking each other. In practice, `ConcurrentHashMap` is more preferred choice because of better performance. It performs both of `get` and `put` operations faster than `Collections.synchronizedMap`

[Report a typo](#)

108 users liked this piece of theory. 3 didn't like it. What about you?



Start practicing

Unskip this topic

[Comments \(13\)](#)

[Useful links \(0\)](#)

[Show discussion](#)

JetBrains Academy

[Tracks](#)

[About](#)

[🕶️ Become beta tester](#)

[Pricing](#)

[Contribute](#)

Be the first to see what's new

[For organizations](#)

[Careers](#)

[Terms](#)

[Support](#)



Made with ❤️ by Hyperskill and JetBrains