**Java Concurrent Collection - ConcurrentHashMap Examples**

Written by  [Nam Ha Minh](https://www.codejava.net/nam-ha-minh)

Last Updated on 13 August 2019   |  [Print](https://www.codejava.net/java-core/concurrency/java-concurrent-collection-concurrenthashmap-examples?tmpl=component&print=1&page=)[Email](https://www.codejava.net/component/mailto/?tmpl=component&template=protostar&link=e25da23e27b7a5f460731c815d4c486cb5de44ec)

In this Java Concurrency tutorial, we will help you understand ConcurrentHashMap - a thread-safe map defined in the java.util.concurrent package. You will be able to use ConcurrentHashMap in your multi-threaded Java programs.

**1. Understanding ConcurrentHashMap**

ConcurrentHashMap is a Map implementation like HashMap and Hashtable, with additional support for concurrency features:

* Unlike Hastable or synchronizedMap which locks the entire map exclusively to gain thread-safety feature, ConcurrentHashMap allows concurrent writer and reader threads. That means it allows some threads to modify the map and other threads to read values from the map at the same time, while Hashtable or synchronizedMap allows only one thread to work on the map at a time. More specifically, ConcurrentHashMap allows any number of concurrent reader threads and a limited number of concurrent writer threads, and both reader and writer threads can operate on the map simultaneously.
  + Reader threads perform retrieval operations such as get, containsKey, size, isEmpty, and iterate over keys set of the map.
  + Writer threads perform update operations such as put and remove.
* Iterators returned by ConcurrentHashMap are weakly consistent, meaning that the iterator may not reflect latest update since it was constructed. An iterator should be used by only one thread and no ConcurrentModificationException will be thrown if the map is modified while the iterator is being used.

ConcurrentHashMap is an implementation of ConcurrentMap which is a subtype of the Map interface. A ConcurrentMap defines the following atomic operations:

* **putIfAbsent(K key, V value)**: associates the specified key to the specified value if the key is not already associated with a value. This method is performed atomically, meaning that no other threads can intervene in the middle of checking absence and association.
* **remove(Object key, Object value)**: removes the entry for a key only if currently mapped to some value. This method is performed atomically.
* **replace(K key, V value)**: replaces the entry for a key only if currently mapped to some value. This method is performed atomically.
* **replace(K key, V oldValue, V newValue)**: replaces the entry for a key only if currently mapped to a given value. This method is performed atomically.

Also note that the methods size() and isEmpty() may return an approximation instead of an exact count due to the concurrent nature of the map. ConcurrentHashMap does not allow null key and null value.

ConcurrentHashMap has such advanced concurrent capabilities because it uses a finer-grained locking mechanism. We don’t delve in to the details of the locking algorithm, but understand that the ConcurrentHashMap uses different locks to lock different parts of the map, which enables concurrent reads and updates.

**2. Java ConcurrentHashMap Example**

The following example demonstrates how ConcurrentHashMap is used in a multi-threaded context. The program creates two writer threads and 5 reader threads working on a shared instance of a ConcurrentHashMap.

The writer thread randomly modifies the map (put and remove). Here’s the code:

public class WriterThread extends Thread {

    private ConcurrentMap<Integer, String> map;

    private Random random;

    private String name;

    public WriterThread(ConcurrentMap<Integer, String> map,

                        String threadName, long randomSeed) {

        this.map = map;

        this.random = new Random(randomSeed);

        this.name = threadName;

    }

    public void run() {

        while (true) {

            Integer key = random.nextInt(10);

            String value = name;

            if(map.putIfAbsent(key, value) == null) {

                long time = System.currentTimeMillis();

                String output = String.format("%d: %s has put [%d => %s]",

                                                time, name, key, value);

                System.out.println(output);

            }

            Integer keyToRemove = random.nextInt(10);

            if (map.remove(keyToRemove, value)) {

                long time = System.currentTimeMillis();

                String output = String.format("%d: %s has removed [%d => %s]",

                                                time, name, keyToRemove, value);

                System.out.println(output);

            }

            try {

                Thread.sleep(500);

            } catch (InterruptedException ex) {

                ex.printStackTrace();

            }

        }

    }

}

The reader thread iterates over each key-value pair in the map and prints it out. Here’s the code:

public class ReaderThread extends Thread {

    private ConcurrentHashMap<Integer, String> map;

    private String name;

    public ReaderThread(ConcurrentHashMap<Integer, String> map, String threadName) {

        this.map = map;

        this.name = threadName;

    }

    public void run() {

        while (true) {

            ConcurrentHashMap.KeySetView<Integer, String> keySetView = map.keySet();

            Iterator<Integer> iterator = keySetView.iterator();

            long time = System.currentTimeMillis();

            String output = time + ": " + name + ": ";

            while (iterator.hasNext()) {

                Integer key = iterator.next();

                String value = map.get(key);

                output += key + "=>" + value + "; ";

            }

            System.out.println(output);

            try {

                Thread.sleep(300);

            } catch (InterruptedException ex) {

                ex.printStackTrace();

            }

        }

    }

}

And the main program creates and starts 2 writer threads and 5 reader threads to work concurrently on a shared instance of a ConcurrentHashMap. Here’s the code:

import java.util.\*;

import java.util.concurrent.\*;

public class ConcurrentHashMapExamples {

    public static void main(String[] args) {

        ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();

        new WriterThread(map, "Writer-1", 1).start();

        new WriterThread(map, "Writer-2", 2).start();

        for (int i = 1; i <= 5; i++) {

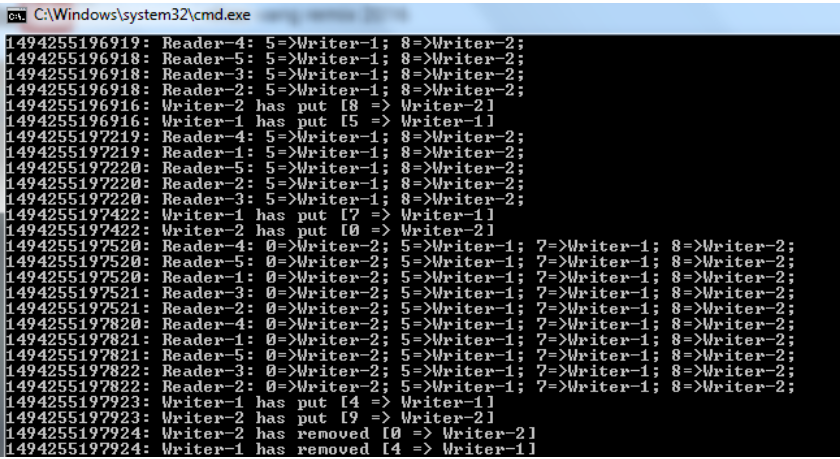
            new ReaderThread(map, "Reader-" + i).start();

        }

    }

}

This program runs forever because all threads run an infinite loop, so you need to press Ctrl + C to stop the program and observe the output. The reader threads let you know that the mp is constantly updated by the writer threads. Here’s a screenshot captured when running the above program on Windows:



**3. Differences between ConcurrentHashMap and HashMap, Hashtable and synchronizedMap**

HashMap is a non-threadsafe Map which should not be used by multiple threads.

Hashtable is a thread-safe Map that allows only one thread to execute a read/update operation at a time.

synchronizedMap is a thread-safe wrapper on a Map implementation. It is generated by the Collections.synchronizedMap(Map)  factory method. A synchronizedMap also allows only a single thread to work on the map at a time.

And ConcurrentHashMap is a thread-safe Map with greater flexibility and higher scalability as it uses a special locking mechanism that enables multiple threads to read/update the map concurrently.

Therefore, you can use ConcurrentHashMap to replace HashMap/Hastable/synchronizedMap for concurrency needs without locking the whole map.

I hope with this understanding, you will be able to decide when, where and how to use ConcurrentHashMapin your Java programs.

**API References:**

* [ConcurrentMap Interface Javadoc](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentMap.html)
* [ConcurrentHashMap Class Javadoc](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html)

**Other Java Concurrent Collections:**

* [Java CopyOnWriteArrayList Examples](https://www.codejava.net/java-core/concurrency/java-concurrent-collection-copyonwritearraylist-examples)
* [Java CopyOnWriteArraySet Examples](https://www.codejava.net/java-core/concurrency/java-concurrent-collection-copyonwritearrayset-example)
* [Java ArrayBlockingQueue Examples](https://www.codejava.net/java-core/concurrency/java-arrayblockingqueue-examples)

**Other Java Concurrency Tutorials:**

* [How to use Threads in Java (create, start, pause, interrupt and join)](https://www.codejava.net/java-core/concurrency/how-to-use-threads-in-java-create-start-pause-interrupt-and-join)
* [Java Synchronization Tutorial](https://www.codejava.net/java-core/concurrency/java-synchronization-tutorial-part-1-the-problems-of-unsynchronized-code)
* [Understanding Deadlock, Livelock and Starvation with Code Examples in Java](https://www.codejava.net/java-core/concurrency/understanding-deadlock-livelock-and-starvation-with-code-examples-in-java)
* [Understanding Java Fork-Join Framework with Examples](https://www.codejava.net/java-core/concurrency/understanding-java-fork-join-framework-with-examples)
* [Understanding Atomic Variables in Java](https://www.codejava.net/java-core/concurrency/understanding-atomic-variables-in-java)