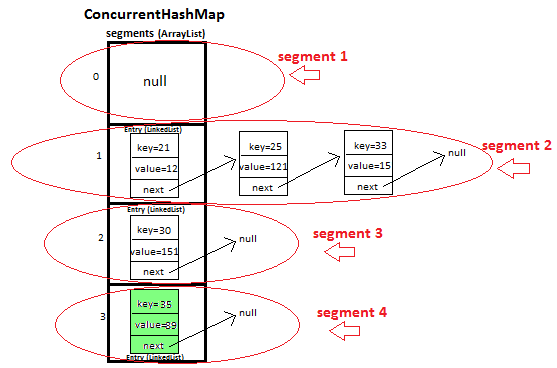
**[Concurrent HashMap in java](http://www.javamadesoeasy.com/2015/04/concurrenthashmap-in-java.html)**



1. *java.util.concurrent.ConcurrentHashMap in java*

java.util.concurrent.**ConcurrentHashMap** is implementation of the java.util.[**Map**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) interface in java.

java.util.concurrent.**ConcurrentHashMap** enables us to store data in key-value pair form. Insertion order of key-value pairs is not maintained. *ConcurrentHashMap* is synchronized in java.

*2) What is hierarchy of ConcurrentHashMap in java?*

-java.lang.Object

-java.util.AbstractMap

 -java.util.concurrent.ConcurrentHashMap

For more detailed hierarchy information read : [**Map hierarchy in java**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html)

*3) Creating java.util.concurrent.ConcurrentHashMap (using* [*constructor*](http://www.javamadesoeasy.com/2015/06/constructor-in-java-constructor.html)*)*

Constructs a new ConcurrentHashMap, Its **initial capacity** is **16**. And **load factor** is **0.75** (We’ll discuss it later in post)

|  |
| --- |
| Map<Integer,String> concurrentHashMap=**new** ConcurrentHashMap<Integer,String>(); |

Defining **ConcurrentHashMap<Integer,String>** means key can of Integer type and value can be String type only, using any other type will cause compilation error.

*4) What is* ***concurrency level*** *in java? What is default concurrency level of java.util.concurrent.ConcurrentHashMap?*

Concurrency level tells how many threads can access ConcurrentHashMap concurrently, default **concurrency level** of ConcurrentHashMap is **16**.

|  |
| --- |
| new ConcurrentHashMap() |

Creates a new ConcurrentHashMap with concurrency level of 16.

*5) How* ***ConcurrentHashMap works****? Can 2 threads on same ConcurrentHashMap object access it concurrently in java?*

*ConcurrentHashMap* is divided into different **segments** based on concurrency level. So different threads can access different **segments** concurrently in java.

**Can threads read the segment of** *ConcurrentHashMap* **locked by some other thread in java?**

Yes. When thread locks one segment for updation it does not block it for retrieval (done by get method) hence some other thread can read the segment (by get method), but it will be able to read the data before locking.

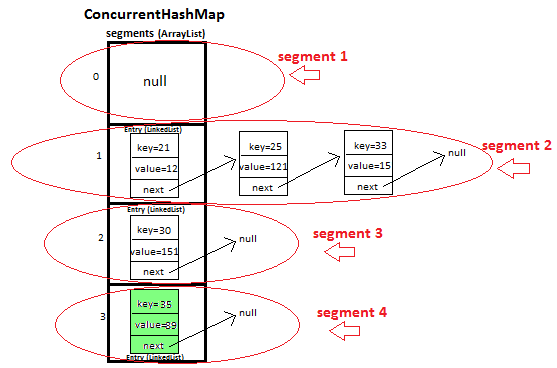
For operations such as putAll concurrent retrievals may reflect removal of only some entries.

For operations such as clear concurrent retrievals may reflect removal of only some entries.

***6****)* ***Segments*** *in ConcurrentHashMap with* ***diagram in java*** *>*

we have ConcurrentHashMap with **4 segments -**

(Diagram shows how **segments** are formed in ConcurrentHashMap)

****

*Now let’s form few questions to clear your doubts (based on above diagram) in java >*

**ConcurrentHashMap Question 1** : What will happen **map.put(25,12)** is called and some other thread concurrently calls **map.get(25)**?

***Answer :*** When **map.put(25,12)** is called **segment 2** will be locked,

**key=25** also lies in **segment 2**, *When thread locks one segment for updation it does not block it for retrieval hence some other thread can read the same segment, but it will be able to read the data before locking* (hence **map.get(25)** will return **121**)

**ConcurrentHashMap Question 2** : What will happen **map.put(25,12)** is called and some other thread concurrently calls **map.get(33)**?

***Answer*** : When **map.put(25,12)** is called **segment 2** will be locked,

**key=33** also lies in **segment 2**, *When thread locks one segment for updation it does not block it for retrieval hence some other thread can read the same segment, but it will be able to read the data before locking* (hence **map.get(33)** will return **15**)

**ConcurrentHashMap Question 3** : What will happen **map.put(25,12)** is called and some other thread concurrently calls **map.put(33,24)**?

***Answer :*** When **map.put(25,12)** is called **segment 2** will be locked,

**key=33** also lies in **segment 2**, *When thread locks one segment for updation it does not allow any other thread to perform updations in same segment until lock is not released on segment*.

hence **map.put(33,24)** will have to wait for **map.put(25,12)** operation to release lock on segment.

**ConcurrentHashMap Question 4** : What will happen **map.put(25,12)** is called and some other thread concurrently calls **map.put(30,29)**?

***Answer :*** When **map.put(25,12)** is called **segment 2** will be locked,

but **key=30** lies in **segment 3**.

*Both the kays lies in different segments,* ***hence both operations can be performed concurrently.***

**ConcurrentHashMap Question 5** : What will happen updations (put/remove) are in process in certain segments and new key-pair have to be put/remove in same segment ?

***Answer :*** When updations are in process *thread locks the segment and it does not allow any other thread to perform updations (put/remove) in same segment until lock is not released on segment*.

*Let’s* ***summarize*** *above section >*

*What operations lock ConcurrentHashMap segment & what operations are allowed when ConcurrentHashMap segment is locked in java >*

* *thread locks one segment for updation (put/remove) & it does not block it for retrieval (get) hence some other thread can read the same segment, but it will be able to read the data before locking*
* *It’s important to know get operations does not lock any segment.*

*7) ConcurrentHashMap* ***putIfAbsent*** *method in java*

*Definition of* ***putIfAbsent*** *method in java >*

|  |
| --- |
| **public** V putIfAbsent(K **key**, V **value**) |

*What do* ***putIfAbsent*** *method do>*

If map does not contain specified **key**, put specified **key-value** pair in map and return null in java.

If map already contains specified **key**, return value corresponding to specified **key**.

***putIfAbsent*** *method is equivalent to writing following code in java >*

|  |
| --- |
| **synchronized** (map){  **if** (!*map*.containsKey(key))  **return** *map*.put(key, value);  **else**  **return** *map*.get(key);     } |

***Program 1*** *to use java.util.concurrent.ConcurrentHashMap’s putIfAbsent method in java >*

|  |
| --- |
| **import** java.util.concurrent.ConcurrentHashMap;  **import** java.util.concurrent.ConcurrentMap;  /\*\* Copyright (c), AnkitMittal JavaMadeSoEasy.com \*/  **public** **class** ConcurrentHashMapTest {  **public** **static** **void** main(String args[]) {        ConcurrentMap<Integer, String> concurrentHashMap =  **new** ConcurrentHashMap<Integer, String>();        concurrentHashMap.put(1, "javaMadeSoEasy");        System.*out*.println("concurrentHashMap : "+concurrentHashMap);          System.*out*.println("\n putIfAbsent method >> "+                            concurrentHashMap.**putIfAbsent**(1, "ankit"));        System.*out*.println("concurrentHashMap : "+concurrentHashMap);        System.*out*.println("\n putIfAbsent method >> "+                            concurrentHashMap.**putIfAbsent**(2, "audi"));        System.*out*.println("concurrentHashMap : "+concurrentHashMap);       }  }  /\*OUTPUT  concurrentHashMap : {1=javaMadeSoEasy}  putIfAbsent method >> javaMadeSoEasy  concurrentHashMap : {1=javaMadeSoEasy}  putIfAbsent method >> null  concurrentHashMap : {2=audi, 1=javaMadeSoEasy}  \*/ |

concurrentHashMap.putIfAbsent(1, "ankit") > returned javaMadeSoEasy because map was already having that key in java.

concurrentHashMap.putIfAbsent(2, "audi") > putted specified key-value pair in map and  returned null because map wasn’t having that key in java.

***Program 2*** *to create method that provides* ***functionality similar to putIfAbsent method of ConcurrentHashMap*** *and to be used with java.util.HashMap in java >*

|  |
| --- |
| **import** java.util.HashMap;  **import** java.util.Map;  /\*\* Copyright (c), AnkitMittal JavaMadeSoEasy.com \*/  **public** **class** HashMapTest {    **static** Map<Integer, String> *map* = **new** HashMap<Integer, String>();    **public** **static** **void** main(String args[]) {  *map*.put(1, "javaMadeSoEasy");            System.*out*.println("hashMap : "+*map*);            System.*out*.println("\n functionalityOfPutIfAbsent method >> "+  ***functionalityOfPutIfAbsent***(1, "ankit"));            System.*out*.println("hashMap : "+*map*);            System.*out*.println("\n functionalityOfPutIfAbsent method >> "+  ***functionalityOfPutIfAbsent***(2, "audi"));            System.*out*.println("hashMap : "+*map*);       }       /\*\*     \* Method is created to be used with HashMap, And     \* method provides functionality similar to putIfAbsent     \* method of ConcurrentHashMap.     \*/  **public** **static** **synchronized** String **functionalityOfPutIfAbsent**(Integer key,String value){  **if (!*map*.containsKey(key))**  **return *map*.put(key, value);**  **else**  **return *map*.get(key);**     }    }  /\*OUTPUT  hashMap : {1=javaMadeSoEasy}  functionalityOfPutIfAbsent method >> javaMadeSoEasy  hashMap : {1=javaMadeSoEasy}  functionalityOfPutIfAbsent method >> null  hashMap : {1=javaMadeSoEasy, 2=audi}  \*/ |

Please note **functionalityOfPutIfAbsent** method is **synchronized,** because this method provides same functionality as that of **ConcurrentHashMap’s putIfAbsent** method and all methods in **ConcurrentHashMap** are **synchronized**.

*functionalityOfPutIfAbsent*(1, "ankit") > returned javaMadeSoEasy because map was already having that key in java.

*functionalityOfPutIfAbsent*(2, "audi") > putted specified key-value pair in map and  returned null because map wasn’t having that key in java.

*8) put element in java.util.concurrent.ConcurrentHashMap*

*put(K key, V value)*

Method allows you put specified *key-value pair* in ConcurrentHashMap. If the map already contains a mapping for the *key*, the old *value* is replaced.

|  |
| --- |
| concurrentHashMap.put(11, "audi"); |

*9) get elements from ConcurrentHashMap in java*

*get(Object key)*

Method returns value corresponding to *key*.

Method returns null if map does not contain *key.*

|  |
| --- |
| concurrentHashMap.get(2); |

Method returns element on 2nd index.

*10) Remove element from ConcurrentHashMap in java*

*remove(Object key)*

Method removes *key*-value pair from ConcurrentHashMap.

|  |
| --- |
| concurrentHashMap.remove(11); |

*11) contains element in ConcurrentHashMap*

*contains(Object object)*

Method returns true if HAshmap contains specified on specified index.

|  |
| --- |
| concurrentHashMap.get(2); |

Method returns element on 2nd index.

*12) Size of java.util.concurrent.ConcurrentHashMap in java*

*size()*

Method returns size of **ConcurrentHashMap.**

|  |
| --- |
| System.*out*.println(concurrentHashMap.size()); |

will print size of concurrentHashMap.

*13) Iterate over java.util.concurrent.ConcurrentHashMap in java*

Before iterating we will put 3 key-value pairs in concurrentHashMap.

      concurrentHashMap.put(11, "audi");

   concurrentHashMap.put(21, "bmw");

   concurrentHashMap.put(31, "ferrari");

13.1) Iterate over keys in java -

*concurrentHashMap.keySet().iterator()* method returns iterator to iterate over keys in ConcurrentHashMap.

|  |
| --- |
| Iterator<Integer> keyIterator=concurrentHashMap.keySet().iterator();  **while**(keyIterator.hasNext()){   System.*out*.println(keyIterator.next());  }  /\*OUTPUT  21  11  31  \*/ |

***Iteration using enhanced for loop in java.***

**concurrentHashMap.keySet()** returns set of keys.

|  |
| --- |
| Set<Integer> keySet=concurrentHashMap.keySet();  **for**(Integer key :keySet){   System.*out*.println(key);  } |

***iterator returned by ConcurrentHashMap over key is*** [***fail-safe***](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html)***.*** **Means any structural modification made to ConcurrentHashMap like adding or removing elements during Iteration will not throw any Exception**.

|  |
| --- |
| Iterator<String> iterator=concurrentHashMap.iterator();  **while**(iterator.hasNext()){   System.*out*.println(iterator.next());   concurrentHashMap.put(4, "d");  } |

key-value has been added (map didn’t contained this key previously) during iteration and no exception is thrown.

13.2) Iterate over values in java -

*concurrentHashMap.values().iterator()* method returns iterator to iterate over keys in ConcurrentHashMap.

|  |
| --- |
| Iterator<String> valueIterator=concurrentHashMap.values().iterator();  **while**(valueIterator.hasNext()){   System.*out*.println(valueIterator.next());  }  /\*OUTPUT  bmw  audi  ferrari  \*/ |

***Iteration using enhanced for loop.***

**concurrentHashMap.values()** returns collection of values.

|  |
| --- |
| Collection<String> collection=concurrentHashMap.values();  **for**(String value :collection){   System.*out*.println(value);  } |

***iterator returned by ConcurrentHashMap over values is*** [***fail-safe***](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html)***.*** **Means any structural modification made to ConcurrentHashMap like adding or removing elements during Iteration will not throw any Exception**.

|  |
| --- |
| Iterator<String> iterator=concurrentHashMap.iterator();  **while**(iterator.hasNext()){   System.*out*.println(iterator.next());   concurrentHashMap.put(5, "d");  } |

key-value has been added (map didn’t contained this key previously) during iteration and no exception is thrown.

13.3) Iterate over Entry in java-

*concurrentHashMap.entrySet().iterator()* method returns iterator to iterate over keys in ConcurrentHashMap in java.

|  |
| --- |
| Iterator<Entry<Integer, String>> entryIterator=concurrentHashMap.entrySet().iterator();  **while**(entryIterator.hasNext()){     System.*out*.println(entryIterator.next());  }  /\*OUTPUT  21=bmw  11=audi  31=ferrari  \*/ |

***Iteration using enhanced for loop.***

**concurrentHashMap.entrySet()** returns collection of values.

|  |
| --- |
| Set<Entry<Integer, String>> entrySet=concurrentHashMap.entrySet();  **for**(Entry<Integer, String> entry:entrySet){        System.*out*.println(entry);  } |

***iterator returned by ConcurrentHashMap over entry is fail-safe.*** **Means any structural modification made to ConcurrentHashMap like adding or removing elements during Iteration will not throw any Exception**.

*14) Some other important methods of java.util.concurrent.ConcurrentHashMap*

**isEmpty()** method returns true if this map contains any key-value pair in java.

**clear()** method removes all key-value pair from map in java.

*15) Complexity of methods in ConcurrentHashMap in java*

|  |  |  |
| --- | --- | --- |
| Operation/ method | **Worst case** | **Best case** |
| *put(K key, V value)* | O(n) | O(1) |
| *get(Object key)* | O(n) | O(1) |

***16****)* ***10 features*** *of java.util.concurrent.ConcurrentHashMap*

1. **ConcurrentHashMap** enables us to store data in key-value pair form in java.

1. **ConcurrentHashMap** is implementation of the java.util.**map** interface in java.

1. **Duplicate key**- ConcurrentHashMap does not allows to store duplicate keys. If the map already contains a mapping for the key, the old value is replaced in java.

1. **Null elements -** ConcurrentHashMap does **not allow to store null key or null value**. Any attempt to store null key or value in ConcurrentHashMap throws runtimeException (NullPointerException).

1. **Insertion order -** ConcurrentHashMap does not maintains insertion order in java.

Example in java-

|  |
| --- |
| Let’s say we add 3 elements in concurrentHashMap  concurrentHashMap.put(1,"ind");  concurrentHashMap.put(2,"aus");  concurrentHashMap.put(3,"sa");  On displaying insertion order will not be maintained i.e.  3,sa  2,aus  1,ind |

1. **synchronized -** ConcurrentHashMap is synchronized in java.

1. **Performance -** ConcurrentHashMap is synchronized, hence its operations are slower as compared to some unSynchronized implementation of map interface in java.

1. **Provides locking in** **segments -** *ConcurrentHashMap* is divided into different **segments** based on concurrency level. So different threads can access different **segments** concurrently in java.

1. ***iterator* are** [***fail-safe***](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html) ***-***
2. *concurrentHashMap.keySet().iterator()*
3. *concurrentHashMap.values().iterator()*
4. *concurrentHashMap.entrySet().iterator()*

all three iterators are ***fail-safe in java.***

1. ***putIfAbsent method is present in ConcurrentHashMap -*** If map does not contain specified **key**, put specified **key-value** pair in map and return null in java.

If map already contains specified **key**, return value corresponding to specified **key**.

*17) When to use java.util.concurrent.ConcurrentHashMap*

1. ConcurrentHashMap can be used when we want to store data in key-value pair form in java.

1. ConcurrentHashMap can be used when we don’t care about insertion order in java.

1. ConcurrentHashMap can be used when we are working in multithreading environment in java.

1. Hashtable is **obsolete in java 5 i.e. JDK 1.5**, hence it is better to use ConcurrentHashMap than using Hashtable in java.

*18) Comparison of performance between HashMap and ConcurrentHashMap*

We will **synchronize HashMap and then compare its performance with ConcurrentHashMap**.

*We can synchronize HashMap by using Collections’s class synchronizedList method in java.*

|  |
| --- |
| *Map synchronizedMap = Collections.synchronizedMap(hashMap);* |

*Now, no 2 threads can access same instance of map concurrently.*

**Hence synchronized HashMap’s performance is slower as compared to ConcurrentHashMap.**

**But why we didn’t compared HashMap (unSynchronized) with ConcurrentHashMap?**

Because performance of unSynchronized collection is always better than some synchronized collection. As, default (unSynchronized) hashMap didn’t cause any locking.

*19) Comparison of performance between Hashtable and ConcurrentHashMap in java*

Hashtable is **obsolete in java 5 i.e. JDK 1.5**, it is better to use ConcurrentHashMap than using Hashtable, because of concurrency level ConcurrentHashMap’s performance is better than Hashtable in java.

*20) What is Load Factor in java?*

Default load factor is 0.75

That means when set will be 75% filled,  it’s capacity will be doubled in java.

Example in java >

Initially when number of elements is 0,  default capacity =16, Load Factor =0.75, ConcurrentHashMap is 0% full in java.

|  |  |  |  |
| --- | --- | --- | --- |
| ***number of elements*** | ***capacity of ConcurrentHashMap*** | ***Load factor*** | ***ConcurrentHashMap filled in %age*** |
| 0 | 16 | 0.75 | 0% |
| 4 | 16 | 0.75 | 25% |
| 8 | 16 | 0.75 | 50% |
| 11 | 16 | 0.75 | 68.7% |

When next element will be added (i.e. 12th element), concurrentHashMap will be 75% filled and capacity will be doubled i.e. from 16 to 32.

|  |  |  |  |
| --- | --- | --- | --- |
| 12 | 32 | 0.75 | 37.5% |

So in this Collection framework tutorial we learned what is java.util.concurrent.ConcurrentHashMap in Collection framework in java