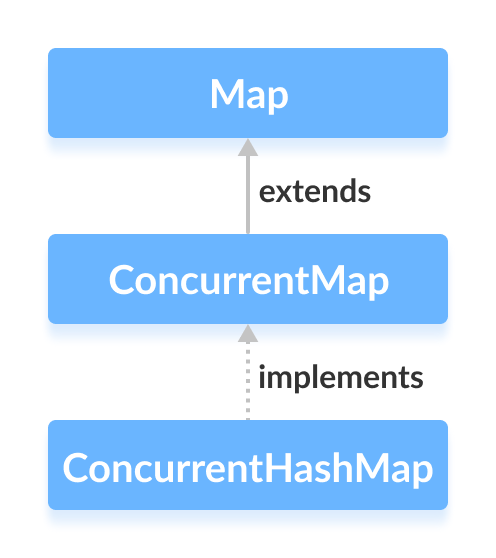
**Java ConcurrentHashMap**

The ConcurrentHashMap class of the Java collections framework provides a thread-safe map. That is, multiple threads can access the map at once without affecting the consistency of entries in a map.

It implements the ConcurrentMap interface.



## Create a ConcurrentHashMap

In order to create a concurrent hashmap, we must import the java.util.concurrent.ConcurrentHashMap package first. Once we import the package, here is how we can create concurrent hashmaps in Java.

// ConcurrentHashMap with capacity 8 and load factor 0.6

ConcurrentHashMap<Key, Value> numbers = new ConcurrentHashMap<>(8, 0.6f);

In the above code, we have created a concurrent hashmap named numbers.

Here,

* Key - a unique identifier used to associate each element (value) in a map
* Value - elements associated by keys in a map

Notice the part new ConcurrentHashMap<>(8, 0.6). Here, the first parameter is **capacity** and the second parameter is **loadFactor**.

* **capacity** - The capacity of this map is 8. Meaning, it can store 8 entries.
* **loadFactor** - The load factor of this map is 0.6. This means, whenever our hash table is filled by 60%, the entries are moved to a new hash table of double the size of the original hash table.

**Default capacity and load factor**

It's possible to create a concurrent hashmap without defining its capacity and load factor. For example,

// ConcurrentHashMap with default capacity and load factor

ConcurrentHashMap<Key, Value> numbers1 = new ConcurrentHashMap<>();

By default,

* the capacity of the map will be 16
* the load factor will be 0.75

## Creating ConcurrentHashMap from Other Maps

Here is how we can create a concurrent hashmap containing all the elements of other maps.

import java.util.concurrent.ConcurrentHashMap;

import java.util.HashMap;

class Main {

public static void main(String[] args) {

// Creating a hashmap of even numbers

HashMap<String, Integer> evenNumbers = new HashMap<>();

evenNumbers.put("Two", 2);

evenNumbers.put("Four", 4);

System.out.println("HashMap: " + evenNumbers);

// Creating a concurrent hashmap from other map

ConcurrentHashMap<String, Integer> numbers = new ConcurrentHashMap<>(evenNumbers);

numbers.put("Three", 3);

System.out.println("ConcurrentHashMap: " + numbers);

}

}

**Output**

HashMap: {Four=4, Two=2}

ConcurrentHashMap: {Four=4, Two=2, Three=3}

## Methods of ConcurrentHashMap

The ConcurrentHashMap class provides methods that allow us to perform various operations on the map.

## Insert Elements to ConcurrentHashMap

* put() - inserts the specified key/value mapping to the map
* putAll() - inserts all the entries from specified map to this map
* putIfAbsent() - inserts the specified key/value mapping to the map if the specified key is not present in the map

For example,

import java.util.concurrent.ConcurrentHashMap;

class Main {

public static void main(String[] args) {

// Creating ConcurrentHashMap of even numbers

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

// Using put()

evenNumbers.put("Two", 2);

evenNumbers.put("Four", 4);

// Using putIfAbsent()

evenNumbers.putIfAbsent("Six", 6);

System.out.println("ConcurrentHashMap of even numbers: " + evenNumbers);

//Creating ConcurrentHashMap of numbers

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

numbers.put("One", 1);

// Using putAll()

numbers.putAll(evenNumbers);

System.out.println("ConcurrentHashMap of numbers: " + numbers);

}

}

**Output**

ConcurrentHashMap of even numbers: {Six=6, Four=4, Two=2}

ConcurrentHashMap of numbers: {Six=6, ONe=1, Four=-4, Two=2}

## Access ConcurrentHashMap Elements

**1. Using entrySet(), keySet() and values()**

* entrySet() - returns a set of all the key/value mapping of the map
* keySet() - returns a set of all the keys of the map
* values() - returns a set of all the values of the map

For example,

import java.util.concurrent.ConcurrentHashMap;

class Main {

public static void main(String[] args) {

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

numbers.put("One", 1);

numbers.put("Two", 2);

numbers.put("Three", 3);

System.out.println("ConcurrentHashMap: " + numbers);

// Using entrySet()

System.out.println("Key/Value mappings: " + numbers.entrySet());

// Using keySet()

System.out.println("Keys: " + numbers.keySet());

// Using values()

System.out.println("Values: " + numbers.values());

}

}

**Output**

ConcurrentHashMap: {One=1, Two=2, Three=3}

Key/Value mappings: [One=1, Two=2, Three=3]

Keys: [One, Two, Three]

Values: [1, 2, 3]

**2. Using get() and getOrDefault()**

* get() - Returns the value associated with the specified key. Returns null if the key is not found.
* getOrDefault() - Returns the value associated with the specified key. Returns the specified default value if the key is not found.

For example,

import java.util.concurrent.ConcurrentHashMap;

class Main {

public static void main(String[] args) {

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

numbers.put("One", 1);

numbers.put("Two", 2);

numbers.put("Three", 3);

System.out.println("ConcurrentHashMap: " + numbers);

// Using get()

int value1 = numbers.get("Three");

System.out.println("Using get(): " + value1);

// Using getOrDefault()

int value2 = numbers.getOrDefault("Five", 5);

System.out.println("Using getOrDefault(): " + value2);

}

}

**Output**

ConcurrentHashMap: {One=1, Two=2, Three=3}

Using get(): 3

Using getOrDefault(): 5

## Remove ConcurrentHashMap Elements

* remove(key) - returns and removes the entry associated with the specified key from the map
* remove(key, value) - removes the entry from the map only if the specified key mapped to the specified value and return a boolean value

For example,

import java.util.concurrent.ConcurrentHashMap;

class Main {

public static void main(String[] args) {

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

numbers.put("One", 1);

numbers.put("Two", 2);

numbers.put("Three", 3);

System.out.println("ConcurrentHashMap: " + numbers);

// remove method with single parameter

int value = numbers.remove("Two");

System.out.println("Removed value: " + value);

// remove method with two parameters

boolean result = numbers.remove("Three", 3);

System.out.println("Is the entry {Three=3} removed? " + result);

System.out.println("Updated ConcurrentHashMap: " + numbers);

}

}

**Output**

ConcurrentHashMap: {One=1, Two=2, Three=3}

Removed value: 2

Is the entry {Three=3} removed? True

Updated ConcurrentHashMap: {One=1}

## Bulk ConcurrentHashMap Operations

The ConcurrentHashMap class provides different bulk operations that can be applied safely to concurrent maps.

### 1. forEach() Method

The forEach() method iterates over our entries and executes the specified function.

It includes two parameters.

* parallelismThreshold - It specifies after how many elements operations in a map are executed in parallel.
* transformer - This will transform the data before the data is passed to the specified function.

For example,

import java.util.concurrent.ConcurrentHashMap;

class Main {

public static void main(String[] args) {

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

numbers.put("One", 1);

numbers.put("Two", 2);

numbers.put("Three", 3);

System.out.println("ConcurrentHashMap: " + numbers);

// forEach() without transformer function

numbers.forEach(4, (k, v) -> System.out.println("key: " + k + " value: " + v));

// forEach() with transformer function

System.out.print("Values are ");

numbers.forEach(4, (k, v) -> v, (v) -> System.out.print(v + ", "));

}

}

**Output**

ConcurrentHashMap: {One = 1, Two = 2, Three = 3}

key: One value: 1

key: Two value: 2

key: Three value: 3

Values are 1, 2, 3,

In the above program, we have used parallel threshold **4**. This means if the map contains 4 entries, the operation will be executed in parallel.

**Variation of forEach() Method**

* forEachEntry() - executes the specified function for each entry
* forEachKey() - executes the specified function for each key
* forEachValue() - executes the specified function for each value

### 2. search() Method

The search() method searches the map based on the specified function and returns the matched entry.

Here, the specified function determines what entry is to be searched.

It also includes an optional parameter parallelThreshold. The parallel threshold specifies after how many elements in the map the operation is executed in parallel.

For example,

import java.util.concurrent.ConcurrentHashMap;

class Main {

public static void main(String[] args) {

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

numbers.put("One", 1);

numbers.put("Two", 2);

numbers.put("Three", 3);

System.out.println("ConcurrentHashMap: " + numbers);

// Using search()

String key = numbers.search(4, (k, v) -> {return v == 3 ? k: null;});

System.out.println("Searched value: " + key);

}

}

**Output**

ConcurrentHashMap: {One=1, Two=2, Three=3}

Searched value: Three

**Variants of search() Method**

* searchEntries() - search function is applied to key/value mappings
* searchKeys() - search function is only applied to the keys
* searchValues() - search function is only applied to the values

### 3. reduce() Method

The reduce() method accumulates (gather together) each entry in a map. This can be used when we need all the entries to perform a common task, like adding all the values of a map.

It includes two parameters.

* parallelismThreshold - It specifies after how many elements, operations in a map are executed in parallel.
* transformer - This will transform the data before the data is passed to the specified function.

For example,

import java.util.concurrent.ConcurrentHashMap;

class Main {

public static void main(String[] args) {

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

numbers.put("One", 1);

numbers.put("Two", 2);

numbers.put("Three", 3);

System.out.println("ConcurrentHashMap: " + numbers);

// Using search()

int sum = numbers.reduce(4, (k, v) -> v, (v1, v2) -> v1 + v2);

System.out.println("Sum of all values: " + sum);

}

}

**Output**

ConcurrentHashMap: {One=1, Two=2, Three=3}

Sum of all values: 6

In the above program, notice the statement

numbers.reduce(4, (k, v) -> v, (v1, v2) -> v1+v2);

Here,

* 4 is a parallel threshold
* (k, v) -> v is a transformer function. It transfers the key/value mappings into values only.
* (v1, v2) -> v1+v2 is a reducer function. It gathers together all the values and adds all values.

**Variants of reduce() Method**

* reduceEntries() - returns the result of gathering all the entries using the specified reducer function
* reduceKeys() - returns the result of gathering all the keys using the specified reducer function
* reduceValues() - returns the result of gathering all the values using the specified reducer function

## ConcurrentHashMap vs HashMap

Here are some of the differences between ConcurrentHashMap and HashMap,

* ConcurrentHashMap is a **thread-safe** collection. That is, multiple threads can access and modify it at the same time.
* ConcurrentHashMap provides methods for bulk operations like forEach(), search() and reduce().

## Why ConcurrentHashMap?

* The ConcurrentHashMap class allows multiple threads to access its entries concurrently.
* By default, the concurrent hashmap is divided into **16 segments**. This is the reason why 16 threads are allowed to concurrently modify the map at the same time. However, any number of threads can access the map at a time.
* The putIfAbsent() method will not override the entry in the map if the specified key already exists.
* It provides its own synchronization.