In this lesson, we discussed how we can make our Map thread-safe by using the `synchronizedMap()` method of the **Collections** class. This method returns **SynchronizedMap** in which all the methods are synchronized. There are lots of differences between a **ConcurrentHashMap** and a **SynchronizedMap**, which we will discuss next.

> Although all the basic operations such as insert, fetch, replace, and remove in a ConcurrentHashMap are similar to the **HashMap**. But we will discuss them again so that you are not confused about how to perform these operations in a **ConcurrentHashMap**.

Differences between a ConcurrentHashMap and SynchronizedMap

Let's discuss some of the differences between a **ConcurrentHashMap** and a **SynchronizedMap**.

1. In a **SynchronizedMap**, the entire Map is locked. So every read/write operation needs to acquire a lock, which makes it very slow. On the other hand in a **ConcurrentHashMap**, only a segment of the Map is locked. Two parallel threads can access or update elements in a different segment, so it performs better.
2. **SynchronizedMap** returns Iterator, which fails fast on concurrent modification. **ConcurrentHashMap** doesn't throw a ConcurrentModificationException if one thread tries to modify it while another is iterating over it.
3. ConcurrentHashMap does not allow null keys or null values while SynchronizedMap allows one null key.

Creating a ConcurrentHashMap

There are four different constructors available to create a ConcurrentHashMap in Java.

Using the no-arg constructor

The simplest way to create a **ConcurrentHashMap** is by using the no-arg constructor. This constructor creates a **ConcurrentHashMap** with an initial capacity of 16 and **load factor** of .75

Below is the code syntax to create a **ConcurrentHashMap**. It states that the key is a String type and the value is an Integer type.

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

Using the constructor that takes initial capacity

We can also provide the initial capacity while creating the **ConcurrentHashMap**. If we are already aware that our **ConcurrentHashMap** will contain more than 16 elements, then it is better to provide some initial capacity as it reduces the number of resizes. In this case, the default load factor (0.75) is also used.

Using the constructor that takes initial capacity and load factor

We can also provide initial capacity with the load factor while creating the **ConcurrentHashMap**. If we don't want frequent resizing, then we can set the load factor to a higher number.

Using the constructor that takes another Map as a parameter

We can also create a **ConcurrentHashMap** using another Map by passing it to the constructor. The newly created **ConcurrentHashMap** will have the same size as that of the passed Map, whereas the load factor will default to **0.75**

Using the constructor that takes initial capacity, load factor, and concurrency level as a parameter

This constructor was used prior to Java 8 and has been kept only for backward compatibility. The concurrency level field is no longer used after Java 8, as the internal implementation has been changed.

Inserting into a ConcurrentHashMap

Let's discuss all the methods that we can use to insert a key-value pair in a **ConcurrentHashMap**.

Using the `put()` method

We can use the `put(K key, V value)` method to insert a key-value pair in the **ConcurrentHashMap**. If the key is not present, then a new key-value pair will be added. If the key is already present, then the value will be updated.

Using the `putIfAbsent()` method

The `putIfAbsent(K key, V value)` method inserts a key-value pair only if it is not already present in the Map. If the key is already present, then its value will not be updated.

Using the `putAll()` method

The `putAll(Map<? extends K, ? extends V> m)` method copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map.

The below example shows the working of **ConcurrentHashMap**.

```
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapDemo {

    public static void main(String args[]) {

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

        stocks.put("Google", 123);
        stocks.put("Microsoft", 654);
        stocks.put("Apple", 345);
        stocks.put("Tesla", 999);

        System.out.println(stocks);

        // Since we are using putIfAbsent(), and Apple is already in the Map, the value will not be added
        stocks.putIfAbsent("Apple", 1000);

        System.out.println(stocks);

    }
}
```

Run Save Reset