Concurrent HashMap

While dealing with Threads in our application HashMap is not a good choice because performance-wise HashMap is not up to the mark.

ConcurrentHashMap is a **thread-safe** implementation of the Map interface in Java, which means multiple threads can access it simultaneously without any synchronization issues

It implements *Serializable*, *ConcurrentMap*<K, V>, *Map*<K, V> interfaces and extends *AbstractMap*<K, V> class.

The underlined data structure for ConcurrentHashMap is **Hashtable**.

At a time any number of threads are applicable for a *read operation* without locking the ConcurrentHashMap object which is not there in HashMap

In ConcurrentHashMap, the Object is divided into a number of segments according to the concurrency level. The default concurrency-level is **16**.

One of the key features of the ConcurrentHashMap is that it provides fine-grained locking, meaning that **it locks only the portion of the map being modified**, rather than the entire map

In ConcurrentHashMap, at a time any number of threads can perform retrieval operation but for *updated* in the object, the thread must lock the particular segment in which the thread wants to operate. This type of locking mechanism is known as **Segment locking** or **bucket locking**. Hence *at a time*, *16 update operations can be performed by threads*.

Inserting null objects is not possible in ConcurrentHashMap as a key or value

Reads can happen very fast while the write is done with a lock on segment level or bucket level.

ConcurrentHashMap *doesn't throw* a **ConcurrentModificationException** if one thread tries to modify it while another is iterating over it

ConcurrentHashMap *doesn't throw* a **ConcurrentModificationException** if one thread tries to modify it, while another is iterating over it

Operations in ConcurrentHashMap

- To insert mappings into a ConcurrentHashMap, we can use **put**() or **putAll**() methods.
- To remove a mapping, we can use **remove(Object key)** method of class ConcurrentHashmap. If the key does not exist in the map, then this function does nothing. To clear the entire map, we can use the **clear()** method.
- We can access the elements of a ConcurrentHashMap using the **get()** method

.....

Fail Fast and Fail Safe Iterators

Concurrent Modification: Concurrent Modification in programming means to modify an object concurrently when another task is already running over it. For example, in Java to modify a collection when another thread is iterating over it. Some Iterator implementations may choose to throw **ConcurrentModificationException** if this behavior is detected.

Fail-Fast iterators immediately throw ConcurrentModificationException if there is structural modification of the collection. Structural modification means adding, removing any element from collection while a thread is iterating over that collection

Iterator on ArrayList, HashMap classes are some examples of fail-fast Iterator.

Fail-Safe iterators don't throw any exceptions if a collection is structurally modified while iterating over it. This is because, they operate on the clone of the collection, not on the original collection and that's why they are called fail-safe iterators.

Iterator on ConcurrentHashMap classes are examples of fail-safe Iterator

How Fail Fast Iterator works ?

Fail-fast iterators use an internal flag called modCount which is updated each time a collection is modified.Fail-fast iterators checks the modCount flag whenever it gets the next value (i.e. using next() method), and if it finds that the modCount has been modified after this iterator has been created, it throws ConcurrentModificationException

If you remove an element via Iterator remove() method, exception will not be thrown. However, in case of removing via a particular collection remove() method, ConcurrentModificationException will be thrown

```
ArrayList<Integer> al = new ArrayList<>();
        //add a few elements to arrayList al
Iterator<Integer> itr = al.iterator();
while (itr.hasNext()) {
        if (itr.next() == 2) {
                itr.remove();   // will not throw Exception
        }
    }
while (itr.hasNext()) {
        if (itr.next() == 3) {
            al.remove(3);   // will throw Exception on next call of next() method
        }
    }
```

Fail Safe Iterator → They make a copy of the internal collection (object array) and iterates over the copied collection.

These iterators require extra memory for cloning. E.g: ConcurrentHashMap

```
ConcurrentHashMap<String, Integer> map   = new ConcurrentHashMap<String, Integer>();
                        // put some key, values to map
Iterator it = map.keySet().iterator();
while (it.hasNext()) {
        String key = (String)it.next();
        System.out.println(key + " : " + map.get(key));
        map.put("SEVEN", 7);
    }
```

The major difference is fail-safe iterator doesn't throw any Exception, contrary to fail-fast Iterator.This is because they work on a clone of Collection instead of the original collection and that's why they are called as the fail-safe iterator.