ConcurrentHashMap

We have HashMap for high performance, and Hashtable if thread safety is required. Then why ConcurrentHashMap ?

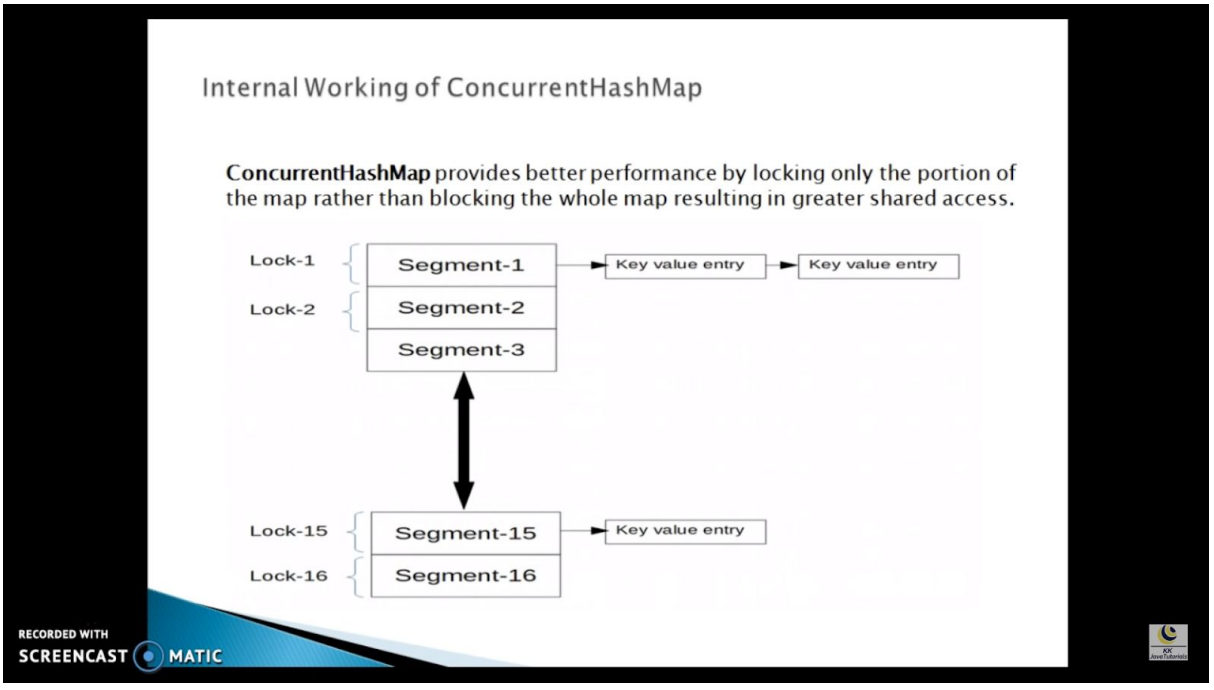
-- Better performance, with a thread safe alternative !

It is a Hashmap, whose operations are thread safe .

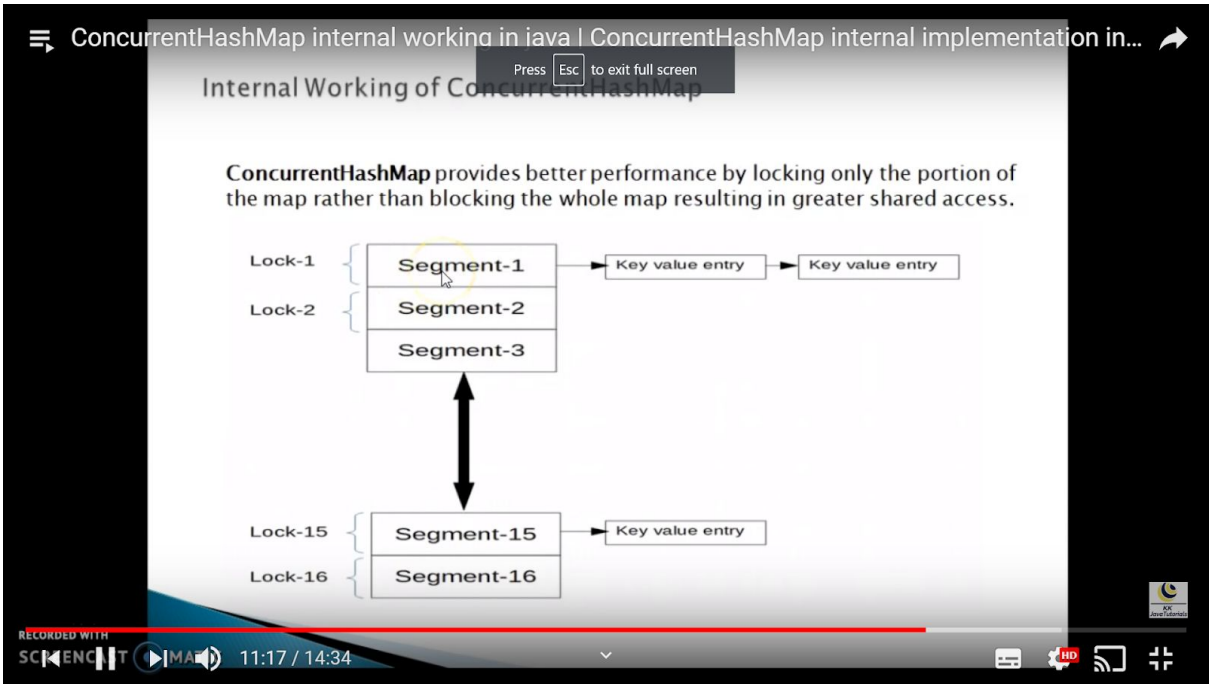
It is also a Hash based Map like Hashmap, but how it differs is the locking strategy.
It has separate locks for separate segments/ buckets, thus locking only a portion of the map.
ConcurrentHashmap uses Reentrant lock for locking mechanism.

Like in Hashmap, there are 16 buckets by default, similarly in ConcurrentHashmap, there are 16 segments, and separate locks for separate segments. So the default concurrency level is 16.
Load factor is 0.75.

Since there are 16 buckets by default, each one having a separate locks, this means that 16 separate threads can operate on these buckets simultaneously, provided the threads are operating on separate segments.
All 16 threads can do operations on the corresponding segments, please note : that 2 threads can write on separate segments (on which they have locks) at the same time, but 2 threads cannot write on the same segment at the same time.
There are 16 segments by default, each having its own lock, so 16 threads can write on those simultaneously.



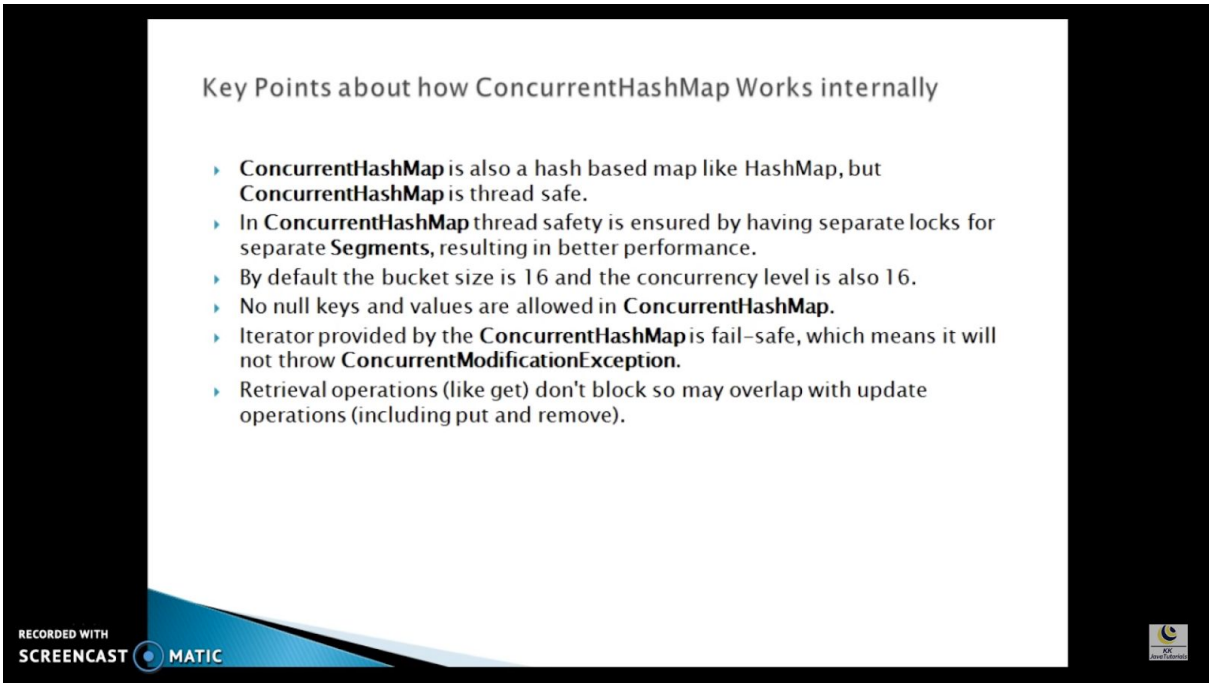
When is ConcurrentHashMap a better choice ?



Like I said above,
2 threads can write on separate segments (on which they have locks) at the same time, but 2 threads cannot write on the same segment at the same time.
But, if one thread is writing on one segment at one time, then another thread can read from it at the same time.

READ AND WRITE may overlap, but 2 writes cannot overlap, on the same segment.

THIS IS THE Reason that while iterating over the Concurrent HashMap, we can add a value, and it is fail-safe.
It does not make copies of the original collection like CopyOnWriteArrayList, but it's every segment is having its own lock, and while adding to the segment, another thread can also read from it, hence it is fail-safe.



Some Q/A :

1. `why and how` we could Synchronize Hashmap?

The Map object is an associative containers that store elements, formed by a combination of a uniquely identify `key` and a mapped `value`. If you have very highly concurrent application in which you may want to modify or read key value in different threads then it's ideal to use Concurrent Hashmap. Best example is Producer Consumer which handles concurrent read/write.

So what does the thread-safe Map means? If `multiple threads` access a hash map concurrently, and at least one of the threads modifies the map structurally, it `must be synchronized externally` to avoid an inconsistent view of the contents.

How?

There are two ways we could synchronized HashMap

1. Java Collections synchronizedMap() method
2. Use ConcurrentHashMap

```
Map<String, String> normalMap = new Hashtable<String,
String>();

//synchronizedMap
synchronizedHashMap = Collections.synchronizedMap(new
HashMap<String, String>());

//ConcurrentHashMap
concurrentHashMap = new ConcurrentHashMap<String,
String>();
```

ConcurrentHashMap

- You should use ConcurrentHashMap when you need very high concurrency in your project.
- It is thread safe without synchronizing the `whole map`.
- Reads can happen very fast while write is done with a lock.
- There is no locking at the object level.
- The locking is at a much finer granularity at a hashmap bucket level.

- ConcurrentHashMap doesn’t throw a `ConcurrentModificationException` if one thread tries to modify it while another is iterating over it.
- ConcurrentHashMap uses multitude of locks.

SynchronizedHashMap

- Synchronization at Object level.
- Every read/write operation needs to acquire lock.
- Locking the entire collection is a performance overhead.
- This essentially gives access to only one thread to the entire map & blocks all the other threads.
- It may cause contention.
- SynchronizedHashMap returns `Iterator`, which fails-fast on concurrent modification.