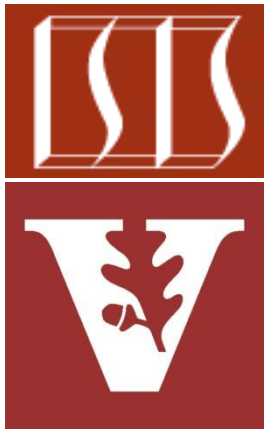
Java Concurrent Collections: ConcurrentHashMap & BlockingQueue



Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Lesson

- Understand the capabilities of Java's concurrent collections
- Recognize the capabilities of Java's ConcurrentHashMap & BlockingQueue

Interface BlockingQueue<E>

Type Parameters:

`E - the type of elements held in this collection`

All Superinterfaces:

`Collection<E>, Iterable<E>, Queue<E>`

All Known Subinterfaces:

`BlockingDeque<E>, TransferQueue<E>`

All Known Implementing Classes:

`ArrayBlockingQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedTransferQueue, PriorityBlockingQueue, SynchronousQueue`

```
public interface BlockingQueue<E>
extends Queue<E>
```

A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

Class ConcurrentHashMap<K,V>

java.lang.Object
  java.util.AbstractMap<K,V>
    java.util.concurrent.ConcurrentHashMap<K,V>

Type Parameters:

`K - the type of keys maintained by this map`

`V - the type of mapped values`

All Implemented Interfaces:

`Serializable, ConcurrentMap<K,V>, Map<K,V>`

```
public class ConcurrentHashMap<K,V>
extends AbstractMap<K,V>
implements ConcurrentMap<K,V>, Serializable
```

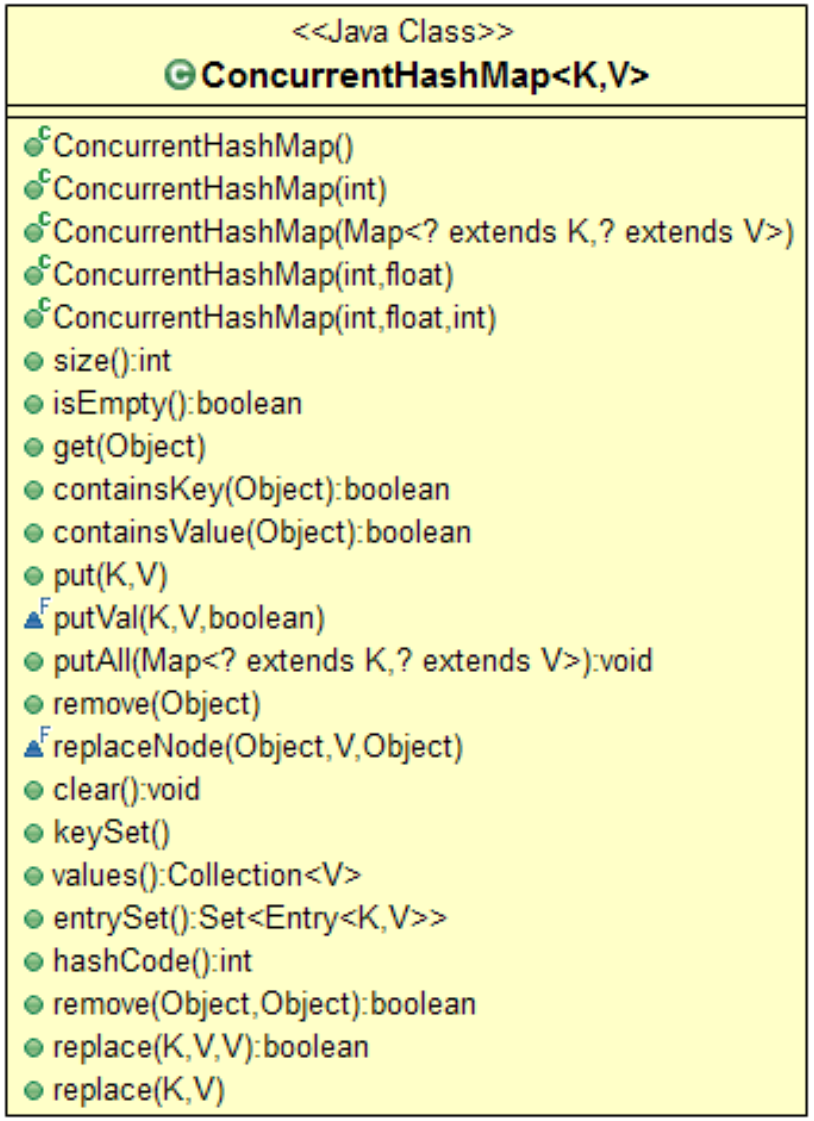
A hash table supporting full concurrency of retrievals and high expected concurrency for updates. This class obeys the same functional specification as `Hashtable`, and includes versions of methods corresponding to each method of `Hashtable`. However, even though all operations are thread-safe, retrieval operations do *not* entail locking, and there is *not* any support for locking the entire table in a way that prevents all access. This class is fully interoperable with `Hashtable` in programs that rely on its thread safety but not on its synchronization details.

Retrieval operations (including `get`) generally do not block, so may overlap with update operations (including `put` and `remove`). Retrievals reflect the results of the most recently *completed* update operations holding upon their onset. (More formally, an update operation for a given key bears a *happens-before* relation with any (non-null) retrieval for that key reporting the updated value.) For aggregate operations such as `putAll` and

Overview of Java ConcurrentHashMap

Overview of Java ConcurrentHashMap

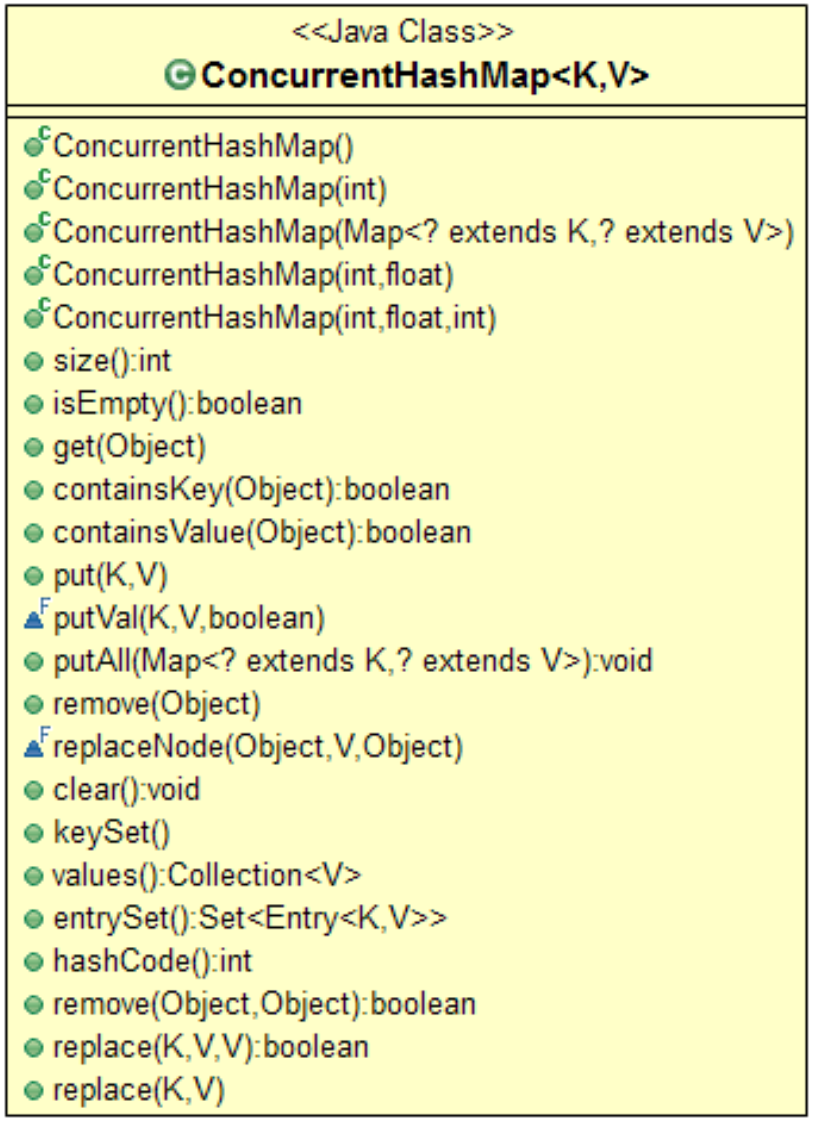
- Provides efficient concurrent operations on key/value pairs via OO & functional programming APIs



See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html

Overview of Java ConcurrentHashMap

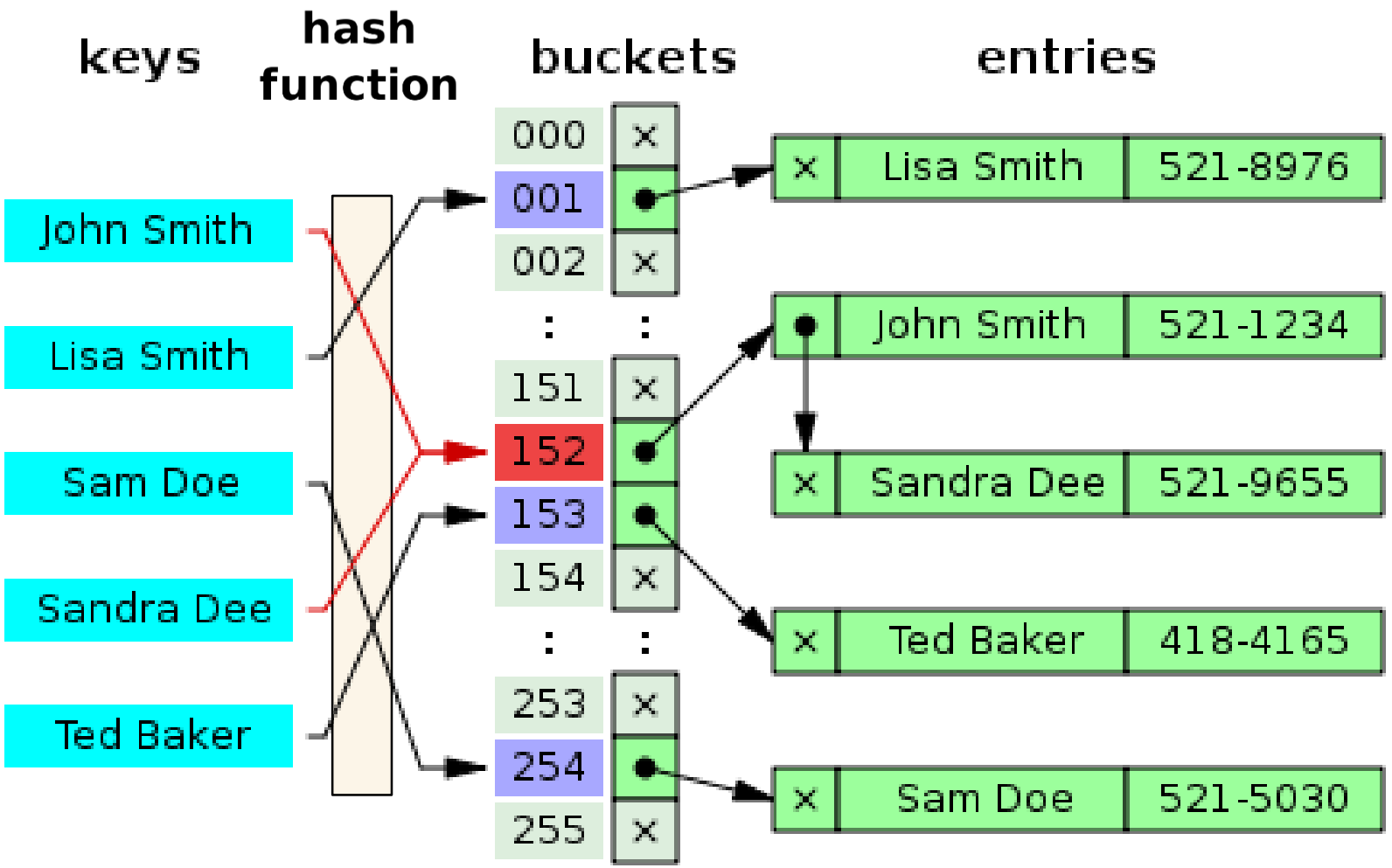
- Provides efficient concurrent operations on key/value pairs via OO & functional programming APIs
  - A highly-optimized "associative array"
    - Cannot contain duplicate keys
      - i.e., each key maps to at most one value



See en.wikipedia.org/wiki/Associative_array

Overview of Java ConcurrentHashMap

- Implemented as a hash table



See en.wikipedia.org/wiki/Hash_table

Overview of Java ConcurrentHashMap

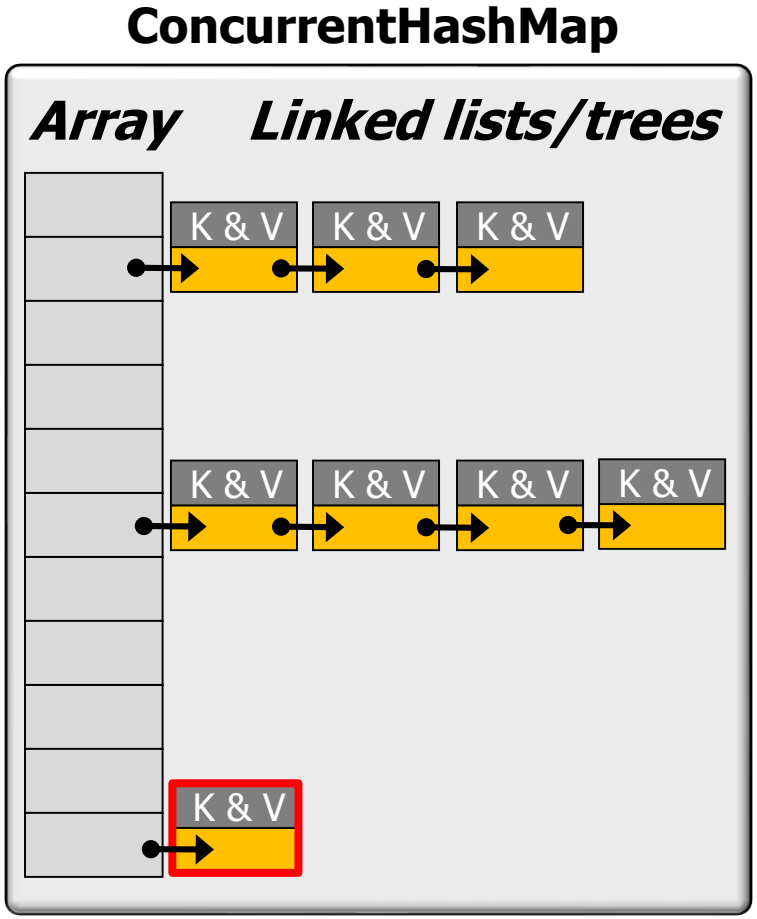
- Implemented as a hash table
  - Insert & retrieve data elements by key

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

initializeMap(map);

// Thread T1
map.put("key1", 42);

// Thread T2
Integer value = map.get("key1");
```



Overview of Java ConcurrentHashMap

- Implemented as a hash table
  - Insert & retrieve data elements by key

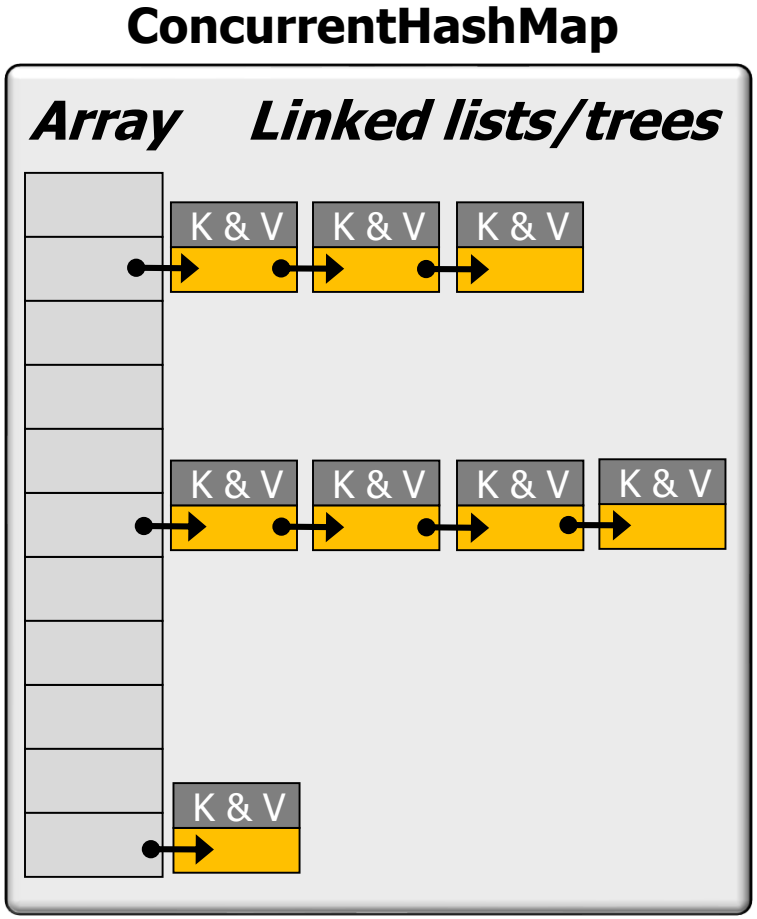
```
Map<String, Integer> map
  = new ConcurrentHashMap<>();

initializeMap(map);

// Thread T1
map.put("key1", 42);

// Thread T2
Integer value = map.get("key1");
```

put() in thread T1 must "happen-before" get() in thread T2

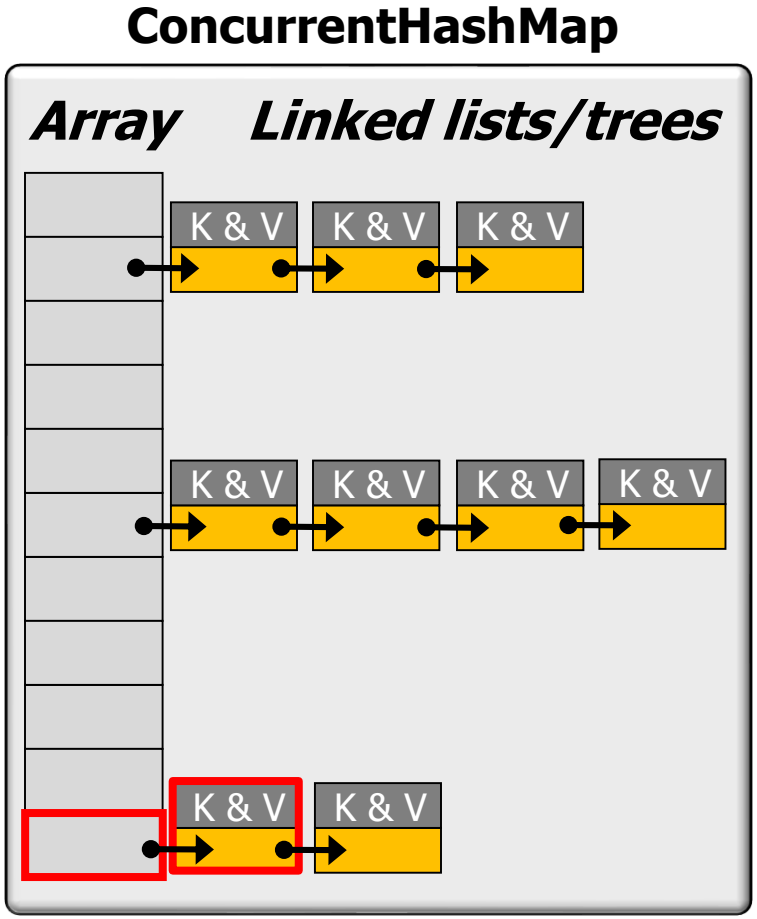


See earlier lesson on "*Java Happens-Before Relationships*"

Overview of Java ConcurrentHashMap

- Implemented as a hash table
  - Insert & retrieve data elements by key
  - Two items that hash to same location in the array are placed in linked list

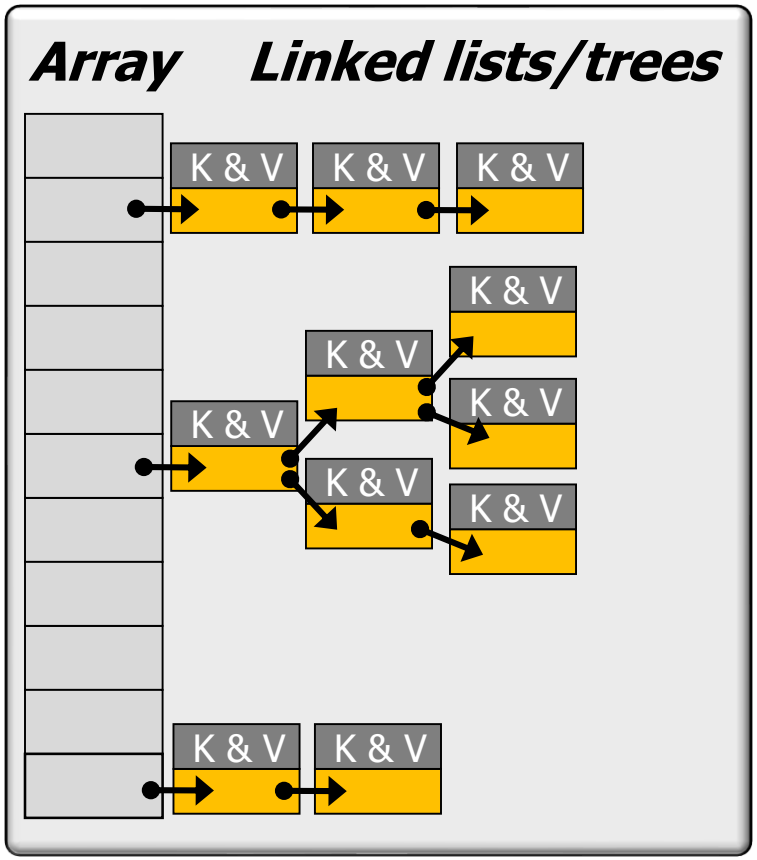
```
map.put("key2", 1066);
```



Overview of Java ConcurrentHashMap

- Implemented as a hash table
  - Insert & retrieve data elements by key
  - Two items that hash to same location in the array are placed in linked list
    - In Java 8+, a linked list is replaced by a binary tree when # of elements in a bucket reaches certain threshold

ConcurrentHashMap



See www.nagarro.com/en/blog/post/24/performance-improvement-for-hashmap-in-java-8

Overview of Java ConcurrentHashMap

- Optimized for multi-core CPUs

Building a better HashMap

How ConcurrentHashMap offers higher concurrency without compromising thread safety

Brian Goetz
Published on August 21, 2003

4

Content series:

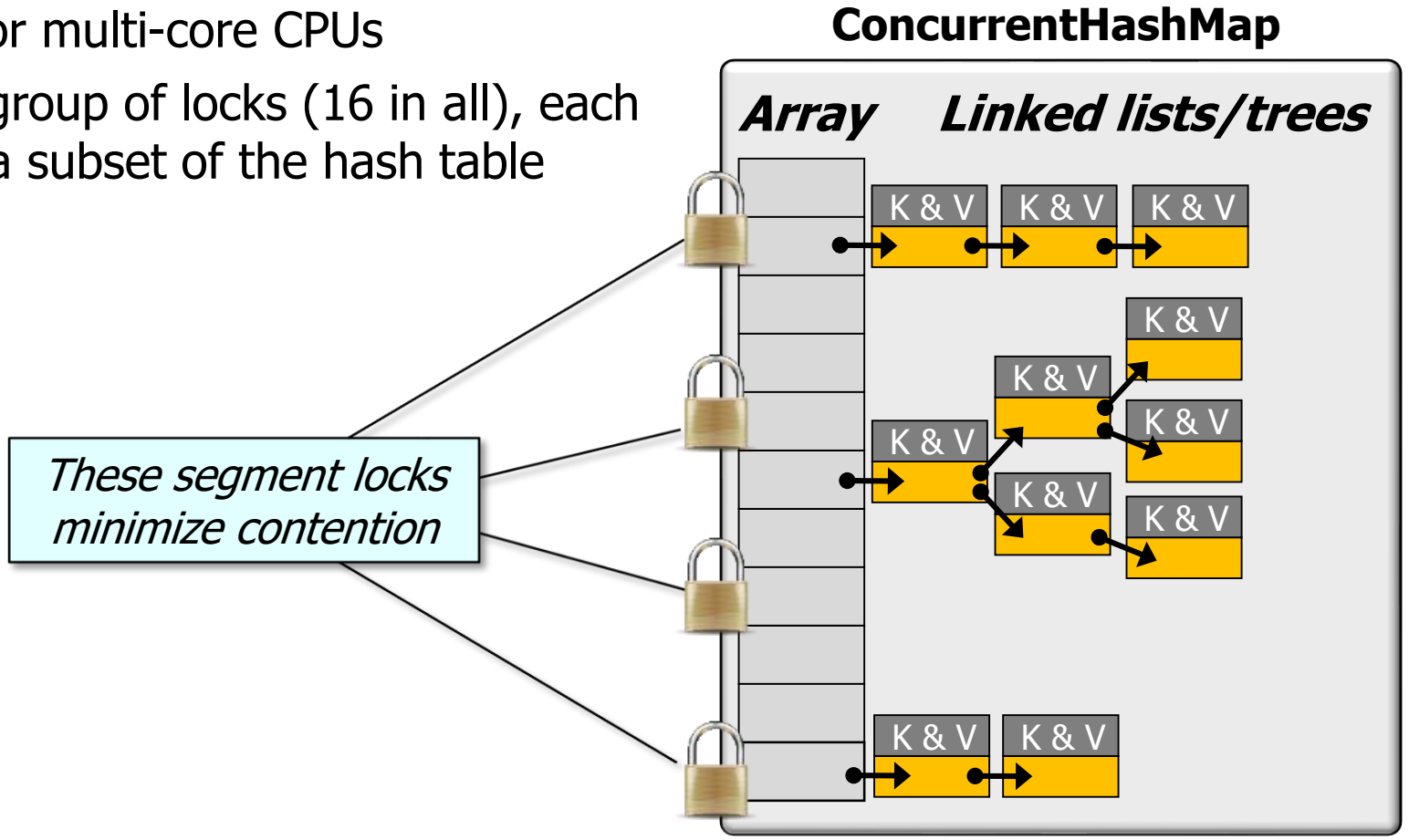
+ This content is part of the series: **Java theory and practice**

In July's installment of *Java theory and practice* ("Concurrent collections classes"), we reviewed scalability bottlenecks and discussed how to achieve higher concurrency and throughput in shared data structures. Sometimes, the best way to learn is to examine the work of the experts, so this month we're going to look at the implementation of `ConcurrentHashMap` from Doug Lea's `util.concurrent` package. A version of `ConcurrentHashMap` optimized for the new Java Memory Model (JMM), which is being specified by JSR 133, will be included in the `java.util.concurrent` package in JDK 1.5; the version in `util.concurrent` has been audited for thread-safety under both the old and new memory models.

See www.ibm.com/developerworks/library/j-jtp08223

Overview of Java ConcurrentHashMap

- Optimized for multi-core CPUs
  - It uses a group of locks (16 in all), each guarding a subset of the hash table



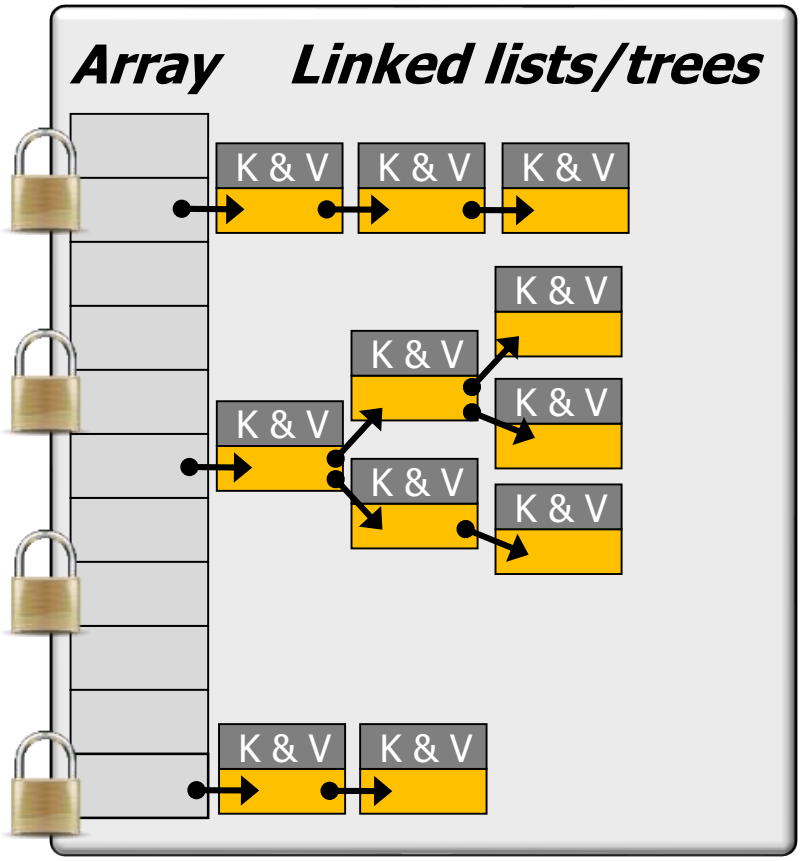
See codepumpkin.com/hashtable-vs-synchronizedmap-vs-concurrenthashmap

Overview of Java ConcurrentHashMap

- Optimized for multi-core CPUs
  - It uses a group of locks (16 in all), each guarding a subset of the hash table

There are common human known uses of this approach!

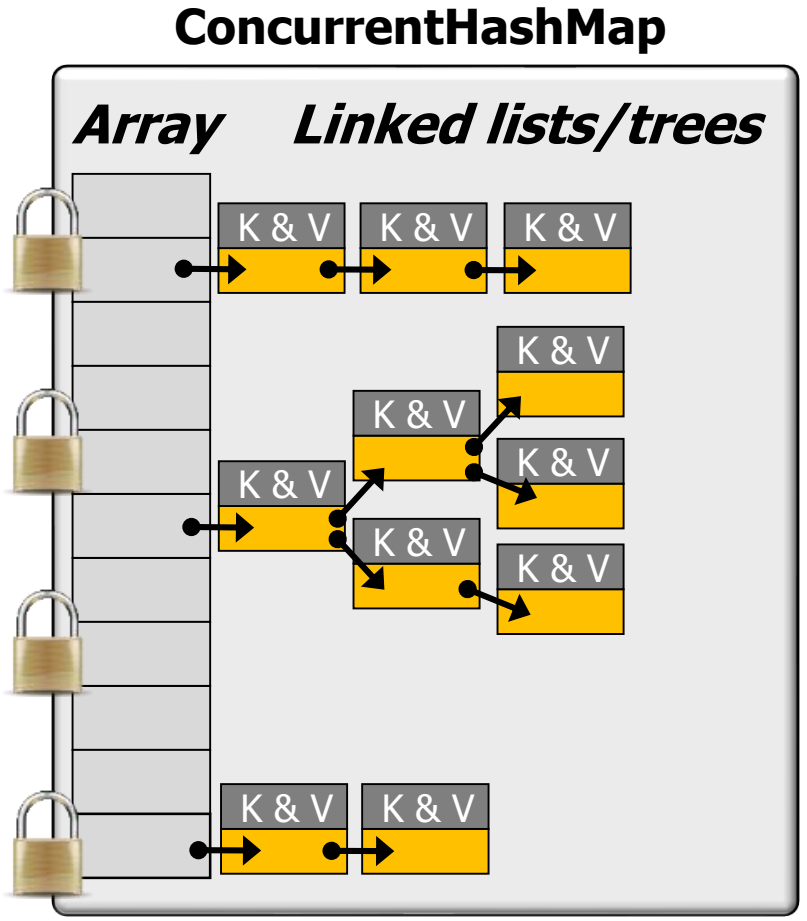
ConcurrentHashMap



See codepumpkin.com/hashtable-vs-synchronizedmap-vs-concurrenthashmap

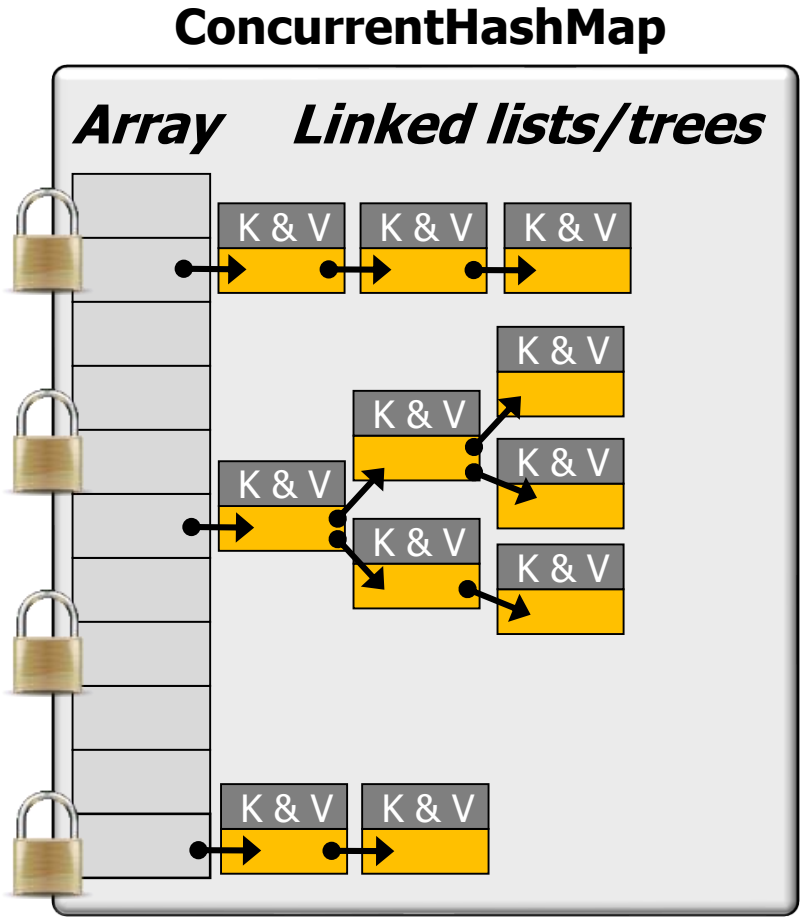
Overview of Java ConcurrentHashMap

- Optimized for multi-core CPUs
  - It uses a group of locks (16 in all), each guarding a subset of the hash table
  - Its methods allow read/write operations with minimal locking



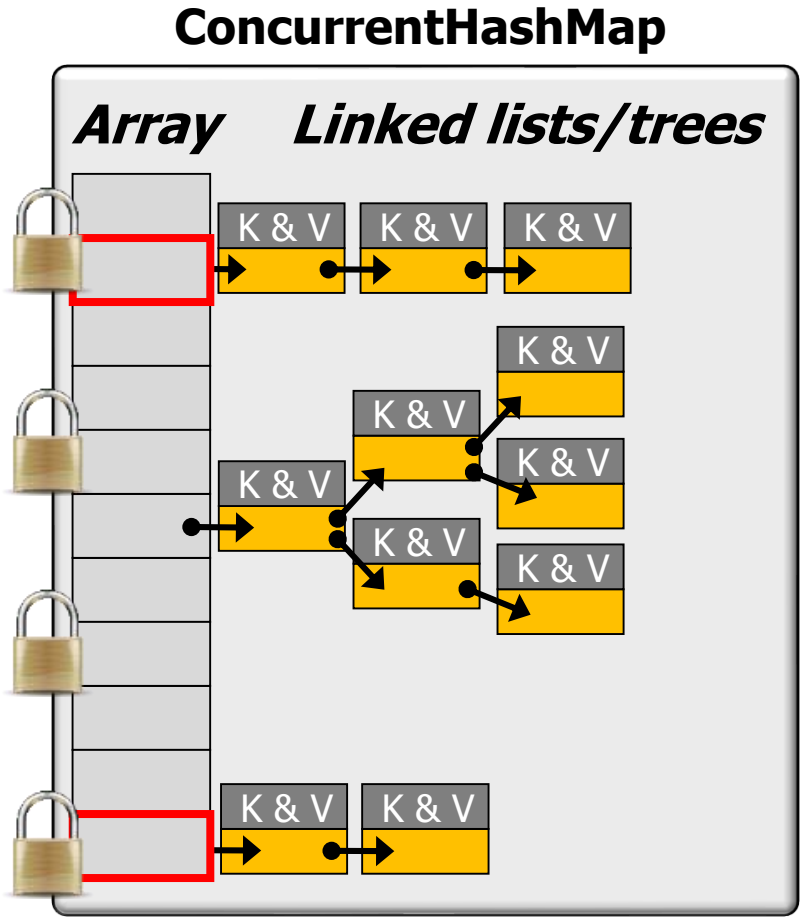
Overview of Java ConcurrentHashMap

- Optimized for multi-core CPUs
  - It uses a group of locks (16 in all), each guarding a subset of the hash table
  - Its methods allow read/write operations with minimal locking, e.g.
    - Reads are concurrent since list cells are immutable (except for data field)



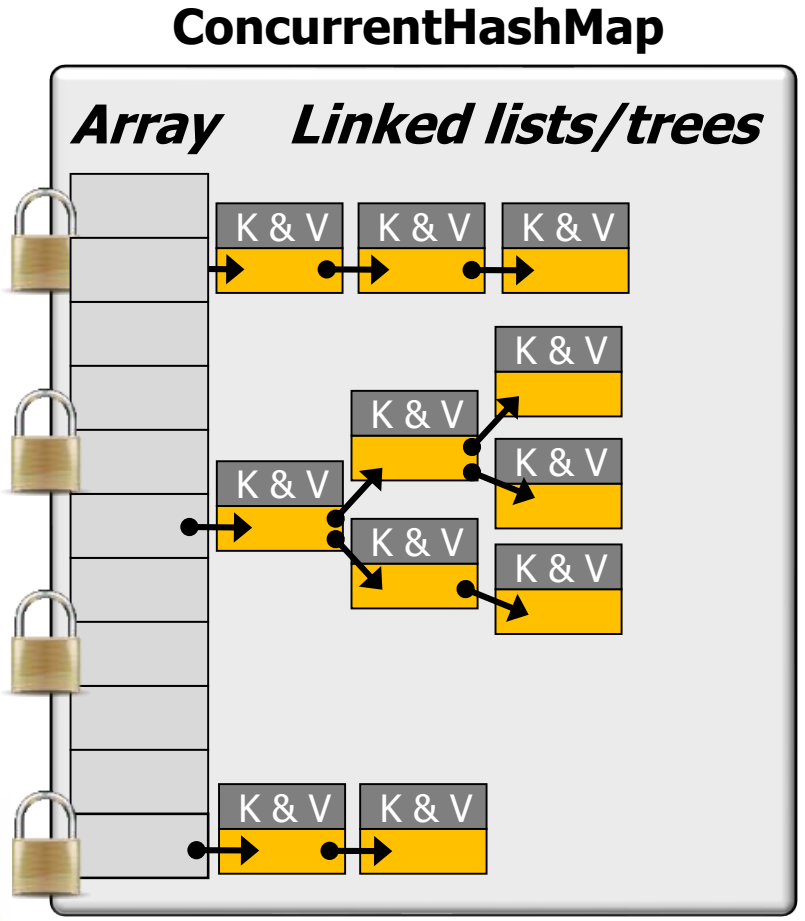
Overview of Java ConcurrentHashMap

- Optimized for multi-core CPUs
  - It uses a group of locks (16 in all), each guarding a subset of the hash table
  - Its methods allow read/write operations with minimal locking, e.g.
    - Reads are concurrent since list cells are immutable (except for data field)
    - Reads & writes are concurrent if they occur in different lists (or trees)



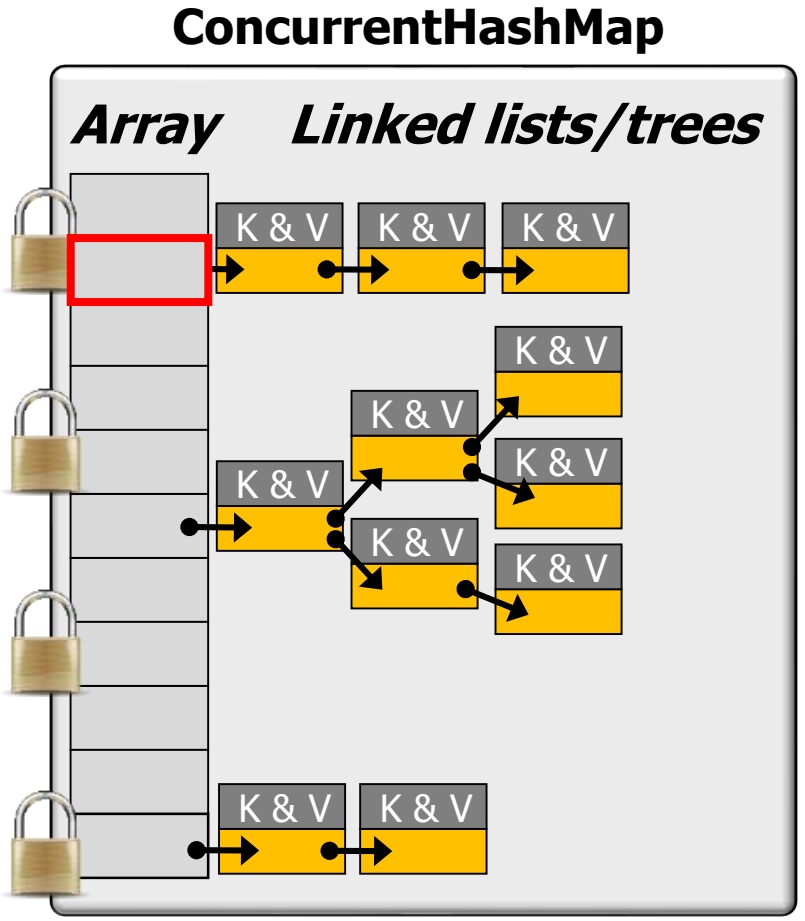
Overview of Java ConcurrentHashMap

- Optimized for multi-core CPUs
  - It uses a group of locks (16 in all), each guarding a subset of the hash table
  - Its methods allow read/write operations with minimal locking, e.g.
    - Reads are concurrent since list cells are immutable (except for data field)
    - Reads & writes are concurrent if they occur in different lists
    - Reads & writes to same list are optimized to avoid locking



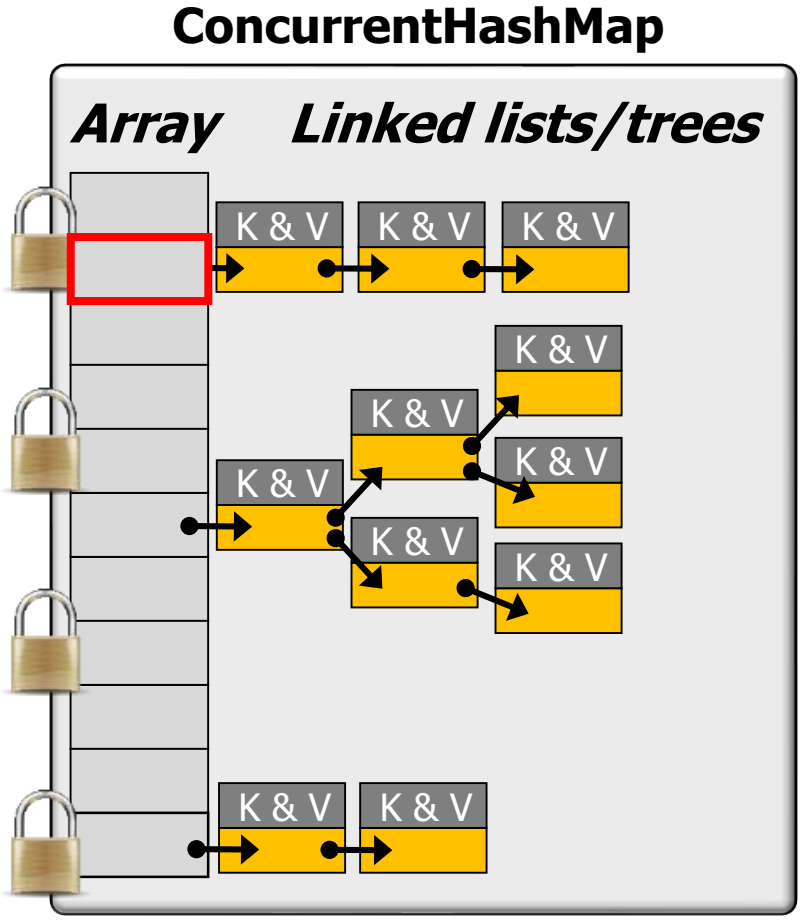
Overview of Java ConcurrentHashMap

- Optimized for multi-core CPUs
  - It uses a group of locks (16 in all), each guarding a subset of the hash table
  - Its methods allow read/write operations with minimal locking, e.g.
    - Reads are concurrent since list cells are immutable (except for data field)
    - Reads & writes are concurrent if they occur in different lists
    - Reads & writes to same list are optimized to avoid locking, e.g.,
      - Atomic add to head of list



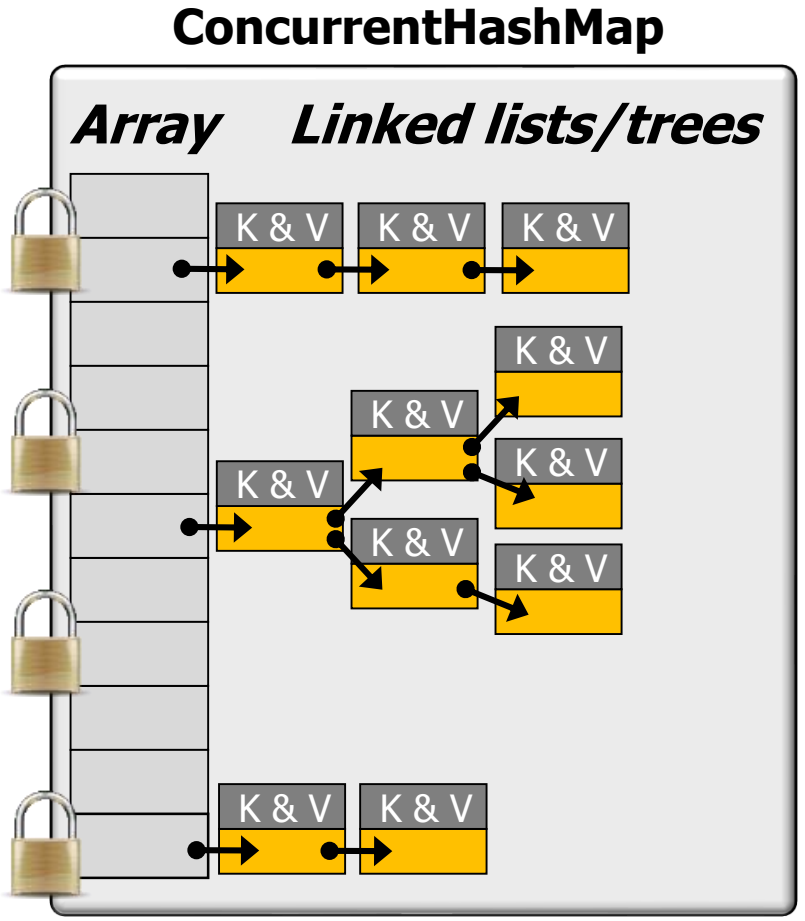
Overview of Java ConcurrentHashMap

- Optimized for multi-core CPUs
  - It uses a group of locks (16 in all), each guarding a subset of the hash table
  - Its methods allow read/write operations with minimal locking, e.g.
    - Reads are concurrent since list cells are immutable (except for data field)
    - Reads & writes are concurrent if they occur in different lists
    - Reads & writes to same list are optimized to avoid locking, e.g.,
      - Atomic add to head of list
      - Remove from list by setting data field to null, rebuild list to skip this cell
        - Unreachable cells are eventually garbage collected



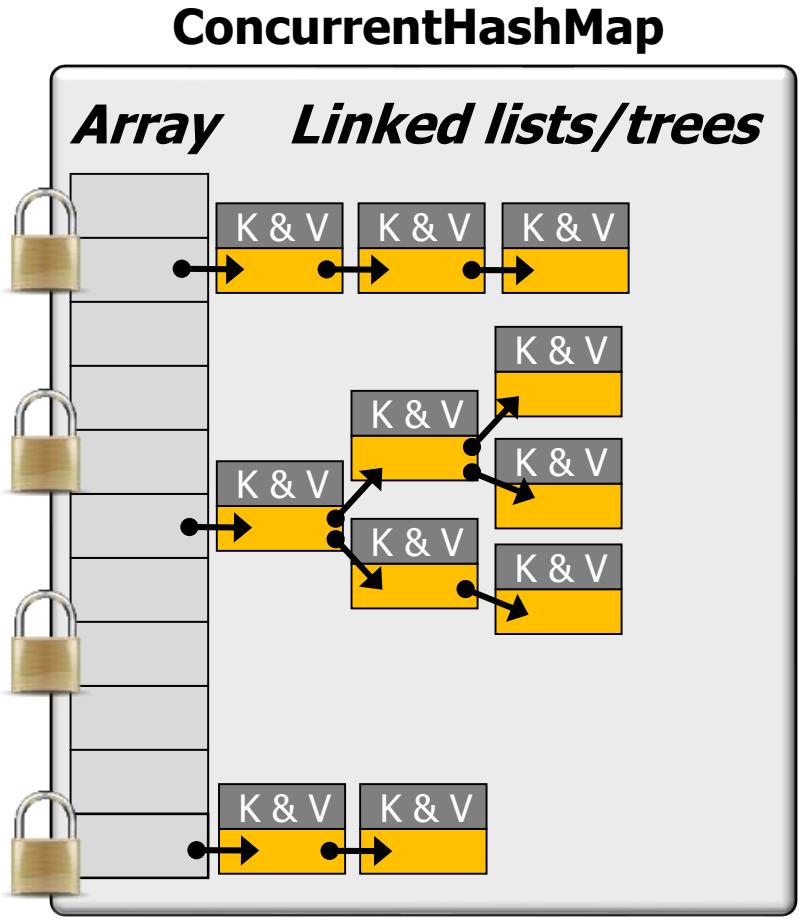
Overview of Java ConcurrentHashMap

- Optimized for multi-core CPUs
  - It uses a group of locks (16 in all), each guarding a subset of the hash table
  - Its methods allow read/write operations with minimal locking, e.g.
    - Reads are concurrent since list cells are immutable (except for data field)
    - Reads & writes are concurrent if they occur in different lists
    - Reads & writes to same list are optimized to avoid locking
    - Can be modified during iteration



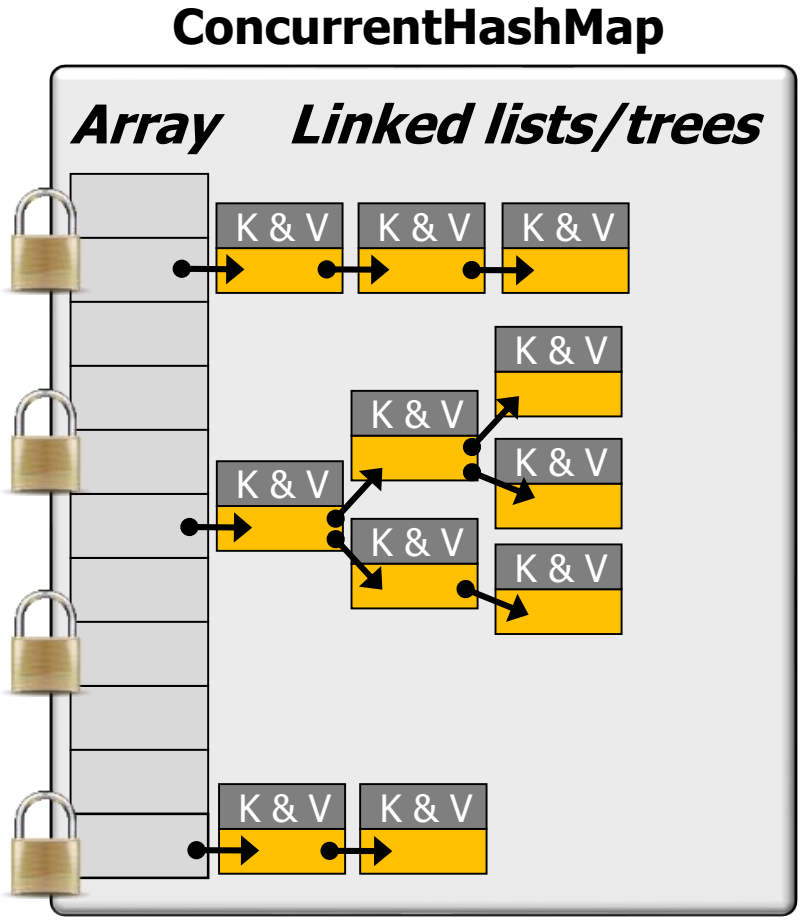
Overview of Java ConcurrentHashMap

- Optimized for multi-core CPUs
  - It uses a group of locks (16 in all), each guarding a subset of the hash table
  - Its methods allow read/write operations with minimal locking, e.g.
    - Reads are concurrent since list cells are immutable (except for data field)
    - Reads & writes are concurrent if they occur in different lists
    - Reads & writes to same list are optimized to avoid locking
    - Can be modified during iteration, e.g.
      - Entire map isn't locked



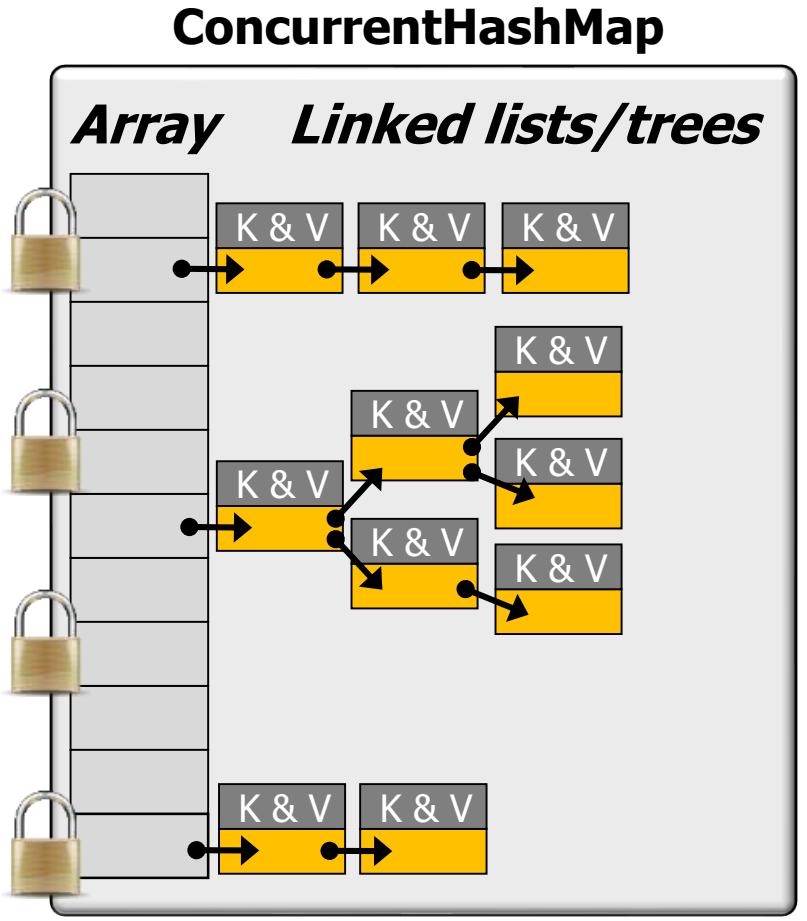
Overview of Java ConcurrentHashMap

- Optimized for multi-core CPUs
  - It uses a group of locks (16 in all), each guarding a subset of the hash table
  - Its methods allow read/write operations with minimal locking, e.g.
    - Reads are concurrent since list cells are immutable (except for data field)
    - Reads & writes are concurrent if they occur in different lists
    - Reads & writes to same list are optimized to avoid locking
    - Can be modified during iteration, e.g.
      - Entire map isn't locked
      - ConcurrentModificationException isn't thrown



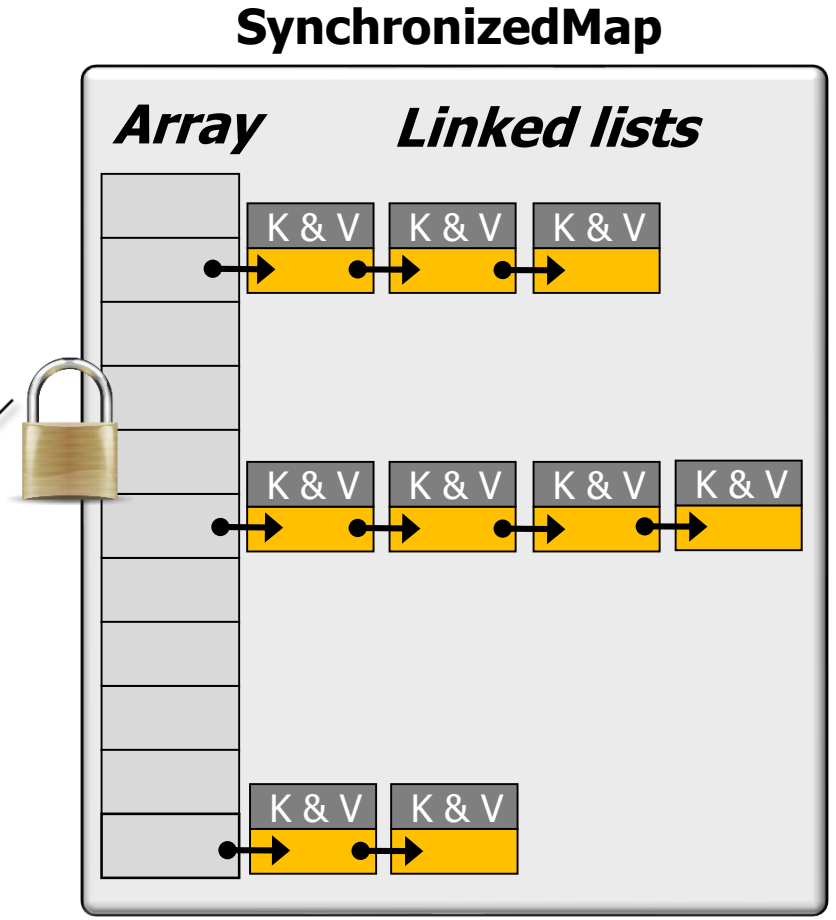
Overview of Java ConcurrentHashMap

- Optimized for multi-core CPUs
  - It uses a group of locks (16 in all), each guarding a subset of the hash table
  - Its methods allow read/write operations with minimal locking, e.g.
    - Reads are concurrent since list cells are immutable (except for data field)
    - Reads & writes are concurrent if they occur in different lists
    - Reads & writes to same list are optimized to avoid locking
  - Can be modified during iteration, e.g.
    - Entire map isn't locked
    - ConcurrentModificationException isn't thrown
    - However, changes may not be visible immediately



Overview of Java ConcurrentHashMap

- Optimized for multi-core CPUs
  - It uses a group of locks (16 in all), each guarding a subset of the hash table
  - Conversely, a SynchronizedMap only uses a single lock



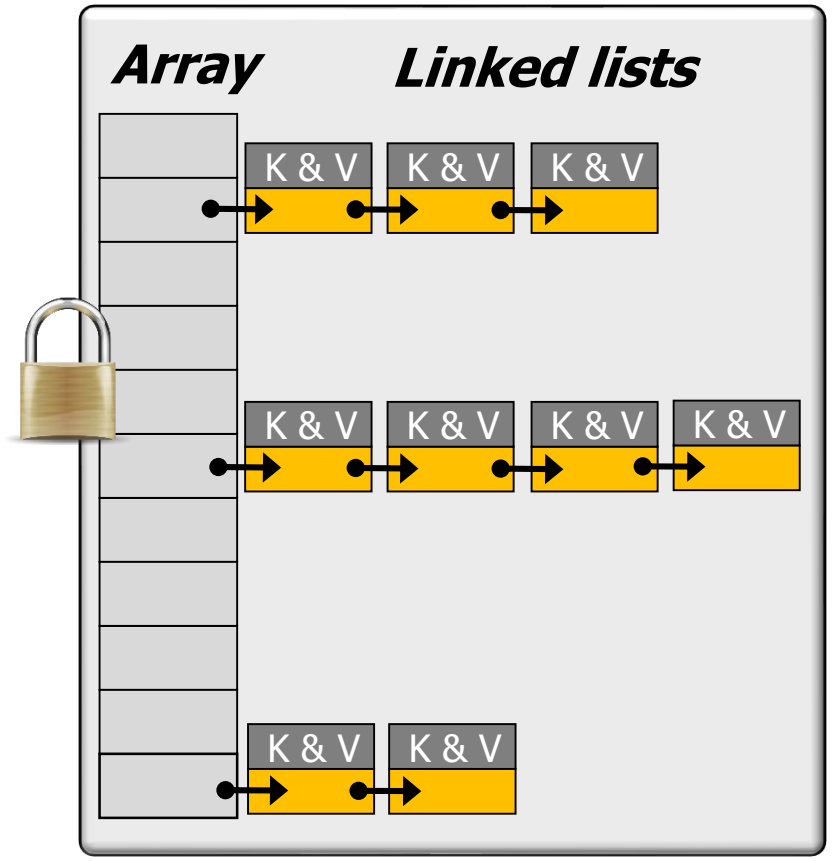
This single lock may cause contention

See codepumpkin.com/hashtable-vs-synchronizedmap-vs-concurrenthashmap

Overview of Java ConcurrentHashMap

- Optimized for multi-core CPUs
  - It uses a group of locks (16 in all), each guarding a subset of the hash table
  - Conversely, a SynchronizedMap only uses a single lock

SynchronizedMap

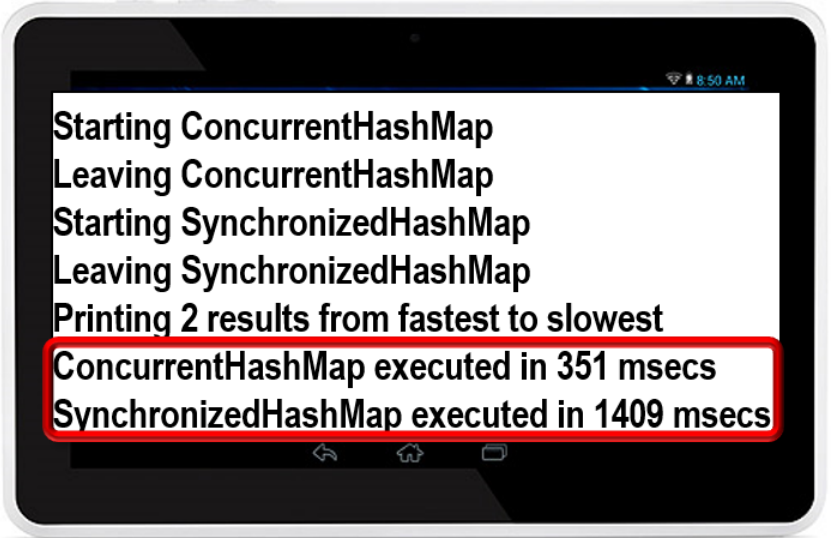


There are also common human known uses of this approach!

See codepumpkin.com/hashtable-vs-synchronizedmap-vs-concurrenthashmap

Overview of Java ConcurrentHashMap

- Optimized for multi-core CPUs
  - It uses a group of locks (16 in all), each guarding a subset of the hash table
  - Conversely, a SynchronizedMap only uses a single lock
  - ConcurrentHashMaps are thus much more scalable than SynchronizedMaps



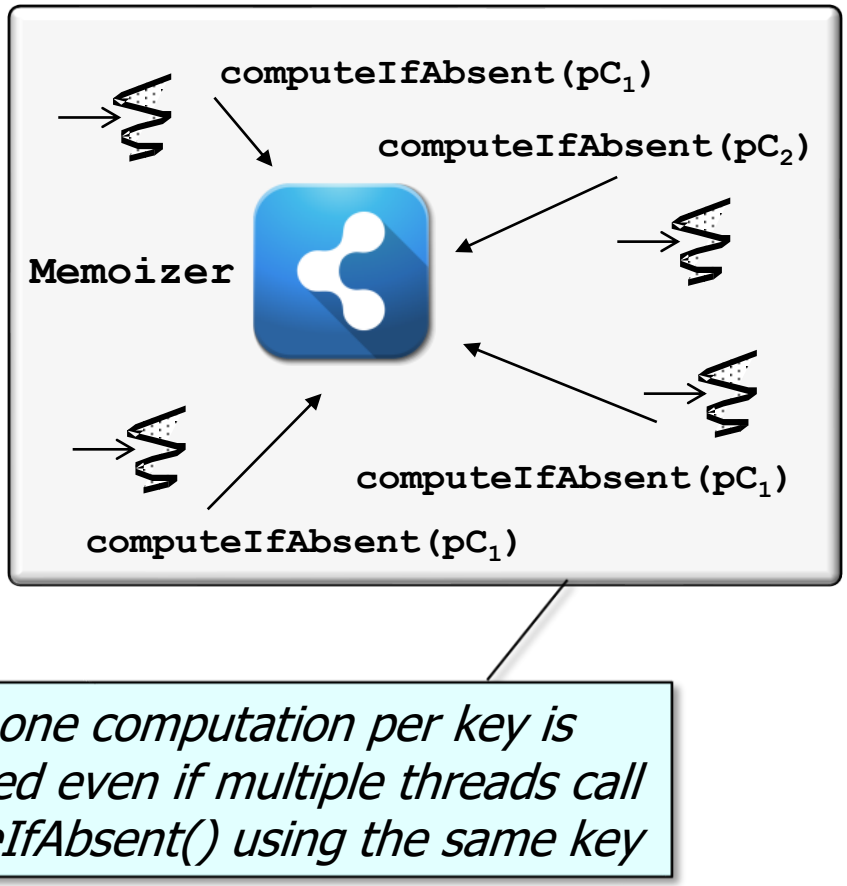

```
public void main(String[] argv){
   ...
   runTest(maxIterations,
           new ConcurrentHashMap());

   runTest(maxIterations,
           Collections.synchronizedMap(new HashMap<>));
   ...
}
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex9

Overview of Java ConcurrentHashMap

- Provides "atomic get-and-maybe-set" methods



Only one computation per key is performed even if multiple threads call computeIfAbsent() using the same key

See dig.cs.illinois.edu/papers/checkThenAct.pdf

Overview of Java ConcurrentHashMap

- Provides "atomic get-and-maybe-set" methods, e.g.
  - If key isn't already associated w/a value, compute its value using the given function & enter it into map

Use

```
return map.computeIfAbsent
   (key,
    k -> mappingFunc(k)));
```

instead of

```
V value = map.get(key);
if (value == null) {
  value =
    mappingFunc.apply(key);
  if (value != null)
    map.put(key, value);
}
return value;
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#computeIfAbsent

Overview of Java ConcurrentHashMap

- Provides "atomic get-and-maybe-set" methods, e.g.
  - If key isn't already associated w/a value, compute its value using the given function & enter it into map
  - If a key isn't already associated w/a value, associate it with the value

Use

```
return map.putIfAbsent
                (key, value);
```

instead of

```
V value = map.get(key);
if (value == null)
  return map.put(key, value);
else
  return value;
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#putIfAbsent

Overview of Java ConcurrentHashMap

- Provides "atomic get-and-maybe-set" methods, e.g.
  - If key isn't already associated w/a value, compute its value using the given function & enter it into map
  - If a key isn't already associated w/a value, associate it with the value
  - Replaces entry for a key only if currently mapped to some value

Use

```
return map.replace(key, value);
```

instead of

```
if (map.containsKey(key))
  return map.put(key, value);
else
  return null;
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#replace

Overview of Java ConcurrentHashMap

- Provides "atomic get-and-maybe-set" methods, e.g.
  - If key isn't already associated w/a value, compute its value using the given function & enter it into map
  - If a key isn't already associated w/a value, associate it with the value
  - Replaces entry for a key only if currently mapped to some value
  - Replaces entry for a key only if currently mapped to given value

Use

```
return map.replace(key,
                   oldValue,
                   newValue);
```

instead of

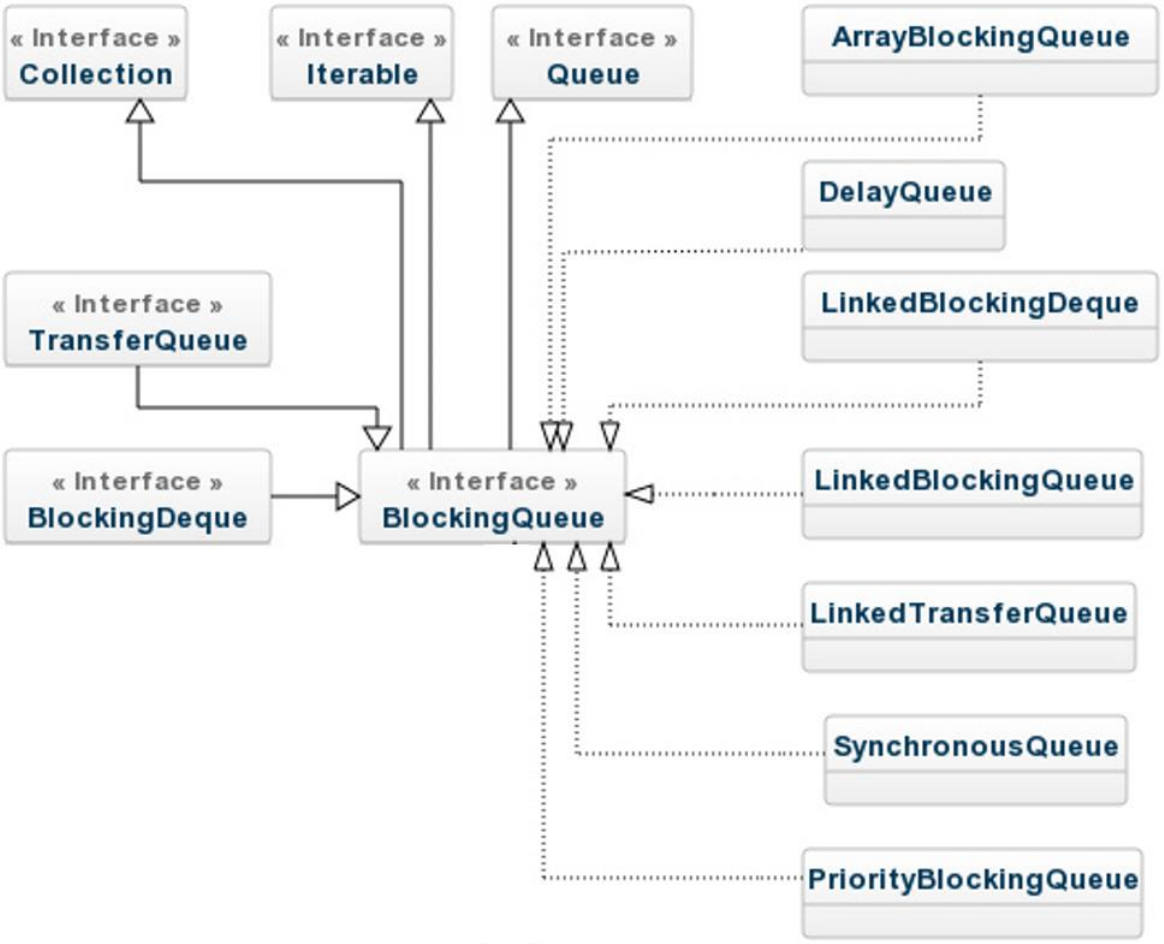
```
if (map.containsKey(key) &&
    Objects.equals(map.get(key),
                   oldValue)) {
   map.put(key, newValue);
   return true;
} else
  return false;
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#replace

Overview of Java BlockingQueue

Overview of Java BlockingQueue

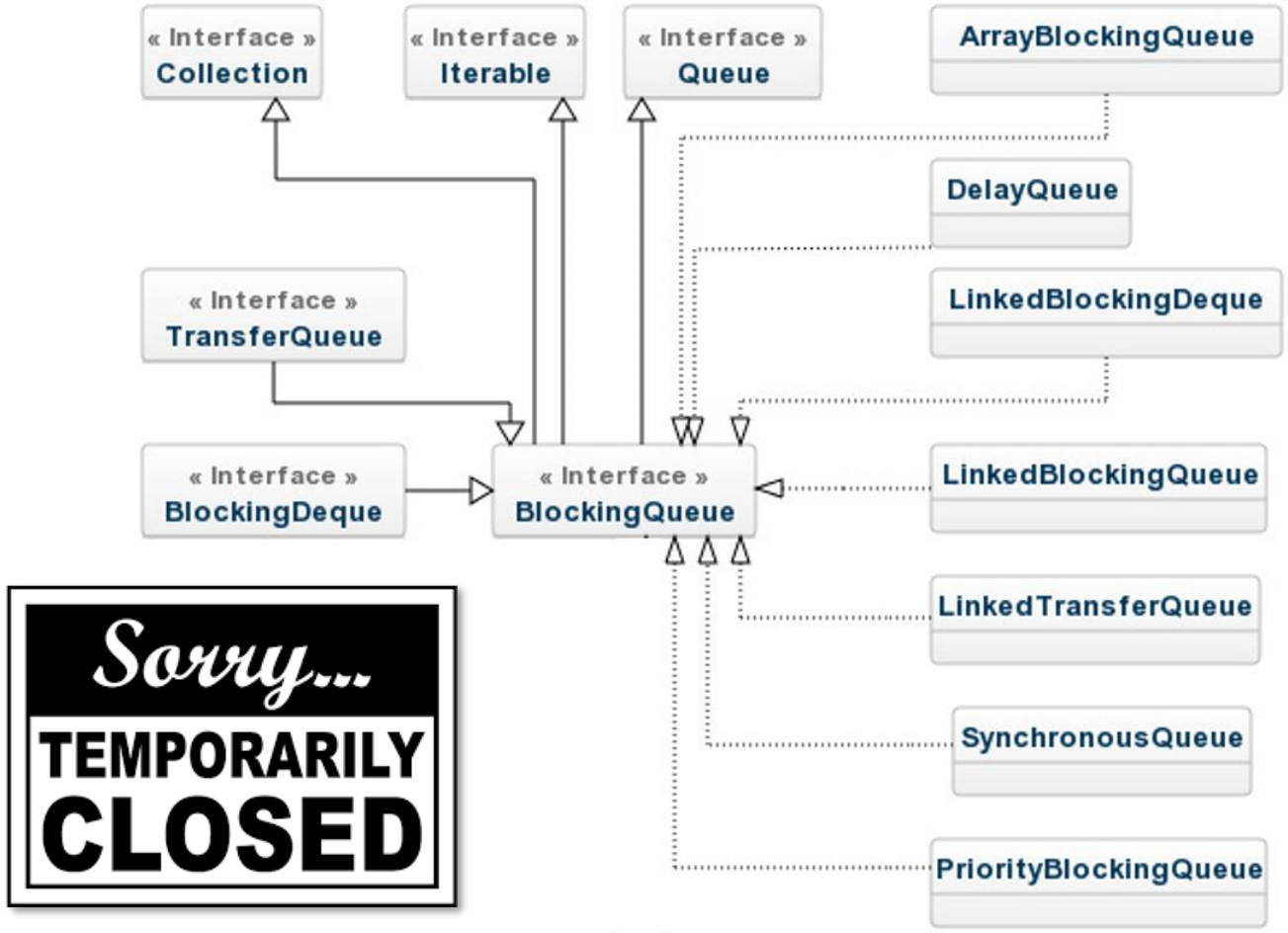
- A Queue supporting operations with certain properties



See docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html

Overview of Java BlockingQueue

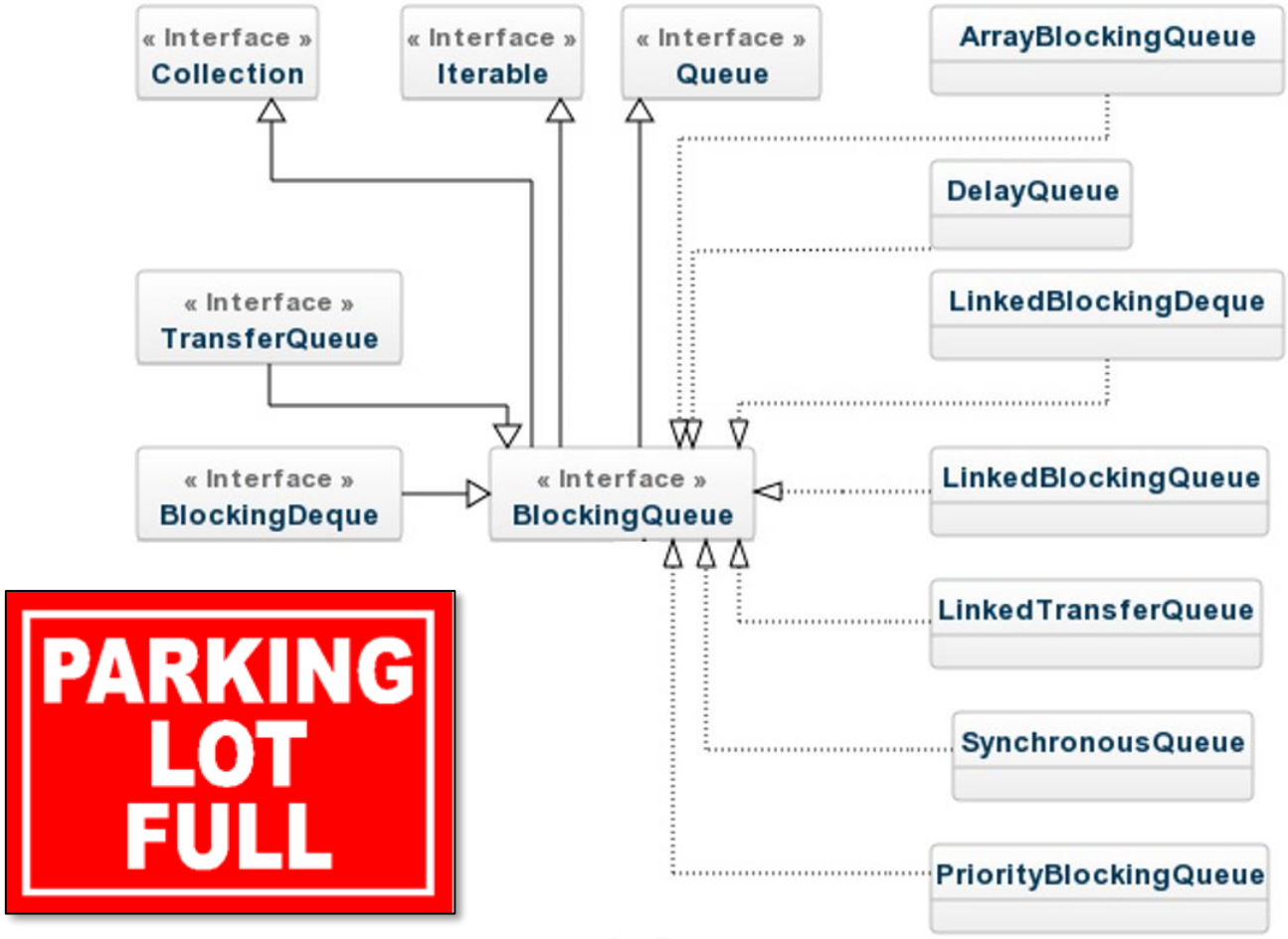
- A Queue supporting operations with certain properties
  - wait for the queue to become non-empty when retrieving an element &



See docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html

Overview of Java BlockingQueue

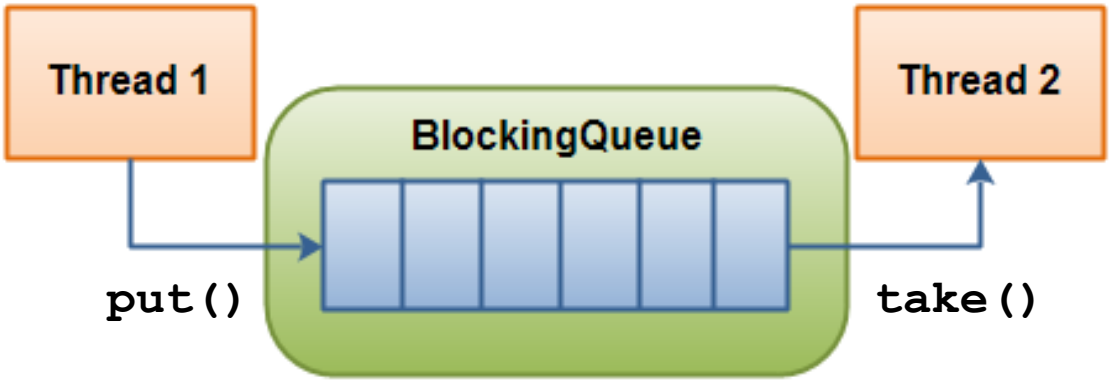
- A Queue supporting operations with certain properties
  - wait for the queue to become non-empty when retrieving an element &
  - wait for space to become available in queue when storing an element



See docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html

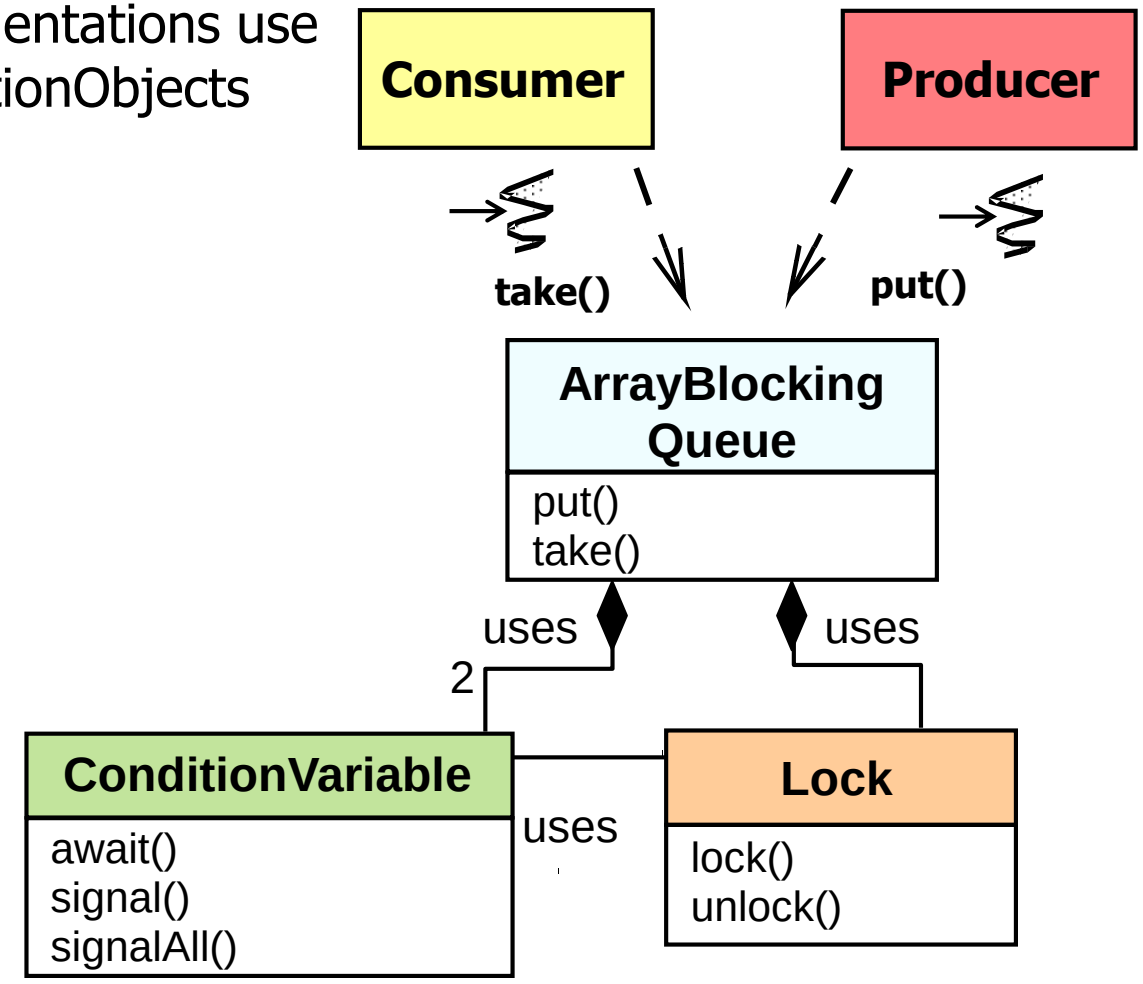
Overview of Java BlockingQueue

- When adding to a full queue or retrieving from an empty queue clients can either block indefinitely, timeout after waiting for a designated time, or poll



Overview of Java BlockingQueue

- Many BlockingQueue implementations use Java ReentrantLock & ConditionObjects



See earlier lessons on "*Java ReentrantLock*" & "*Java ConditionObject*" for examples

End of Java Concurrent Collections: ConcurrentHash Map & BlockingQueue