Core Java 8

Concurrent Collections In Java

Lesson Objectives

After completing this lesson, participants will be able to:

- Understand Concurrent Collections in java
- Use Queue ,BlockingQueue ,ConcurrentMap Interface API
- Implement parallel Search on ConcurrentHashMap

Implementing Concurrency at the API Level

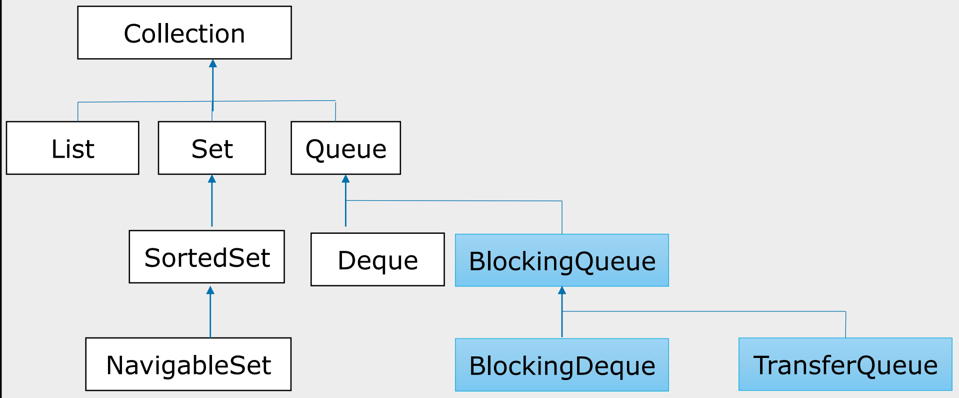
Most of the Collections classes objects (like ArrayList, LinkedList, HashMap etc) are non-synchronized in nature i.e. multiple threads can perform on a object at a time simultaneously. Therefore objects are not thread-safe.

Very few Classes objects (like Vector, Stack, HashTable) are synchronized in nature i.e. at a time only one thread can perform on an Object. But here the problem is performance is low because at a time single thread execute an object and rest thread has to wait.

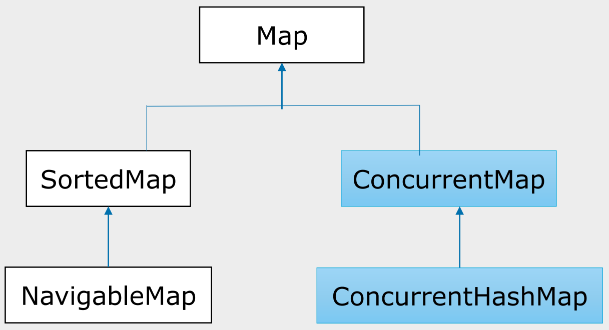
The main problem is when one thread is iterating an Collections object then if another thread cant modify the content of the object. If another thread try to modify the content of object then we will get RuntimeException saying ConcurrentModificationException.

Because of the above reason Collections classes is not suitable or we can say that good choice for Multi-threaded applications.

Hierarchy of Collection and Map, Concurrent Interfaces



Hierarchy of Collection and Map, Concurrent Interfaces



What Does It Mean for an Interface to Be Concurrent?

Concurrent interfaces define the contract in concurrent environment.

JDK provides the implementations that follow these contracts.

If we want to implement those interfaces ,we must follow that contract.

Since Concurrency is complex – Dealing with 10 threads is not same as dealing with 1000 thread.

So Different implementations are required.

Why You Should Avoid Vectors and Stacks

Vector And Stack – Thread Safe Structures

These are legacy classes ,very poorly implemented.

These classes should not be used.

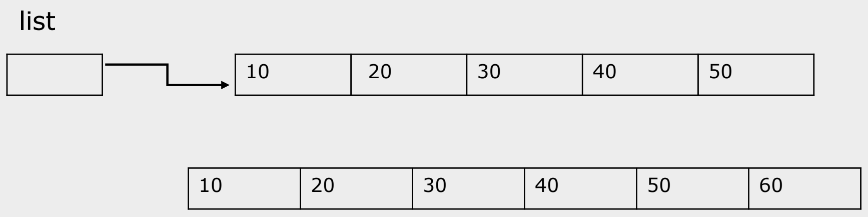
Understanding Copy On Write Arrays

Copy on Write exists for list and set

No locking for read operations

Write Operations create a new structure

The new structure then replaces the previous one.



All the threads can perform read operations on this array parallel and freely . While adding new elements in this array , it will create new copy internally and add the new element in that copy . All the threads which are reading old copy freely without seeing this new array .When the new array is ready ,the pointer is moved from old array to new array in synchronized manner so new read operations will see this new structure while any threads iterating old array will not see this new modification.

Introducing Queue and Deque, and Their Implementations

Queue :The Queue is used to insert elements at the end of the queue and removes from the beginning of the queue. It follows FIFO concept.

Deque : The java.util.Deque interface is a subtype of the java.util.Queue interface. The Deque is related to the double-ended queue that supports addition or removal of elements from either end of the data structure, it can be used as a queue (first-in-first-out/FIFO) or as a stack (last-in-first-out/LIFO). These are faster than Stack and LinkedList.

ArrayBlockingQueue : It is a bounded blocking queue built on an array. Blocking means once the queue is full it will not extends itself. Adding new elements will not be possible then.

ConcurrentLinkedQueue : It is an unbounded blocking queue where we can add as many elements as we want

Understanding How Queue Works in a Concurrent Environment

Producer will add the elements in this queue and Consumer will consume the elements from this queue.

10	34	20	239	23	

But in concurrent environment there will be many producer and consumers working on this queue. Each of them in its own thread. A thread does not know how many elements are there in the queue.

Issues with this implementation :
1.What happens when the queue is full and we want to add new elements to it?
2. What happens when the queue is empty and we need to get the element from it ?

Adding Elements to a Queue That Is Full: How Can It Fail?

Consider the below queue(ArrayBlockingQueue) which is already full

We want to add new element here

20	49	23	24	23	24

boolean add(Element) : //fail : throws IllegalArguementException

boolean offer(Element) : //fail : return false instead of throwing exception

void put(Element): //blocks until a cell becomes available

boolean offer(Element,timeout,timeunit) : //want to add the element in queue and ready to wait given time duration . Past that time period ,it fails by returning false.

Here we are considering the ArrayBlockingQueue .
If we are working with ConcurrentLinkedListQueue this issue will not arise as it will adjust the size for new elements.

For Deque :
Deque can accept elements at the head of a queue :
addFirst(),offerFirst(),
And for BlockingDeque :putFirst()

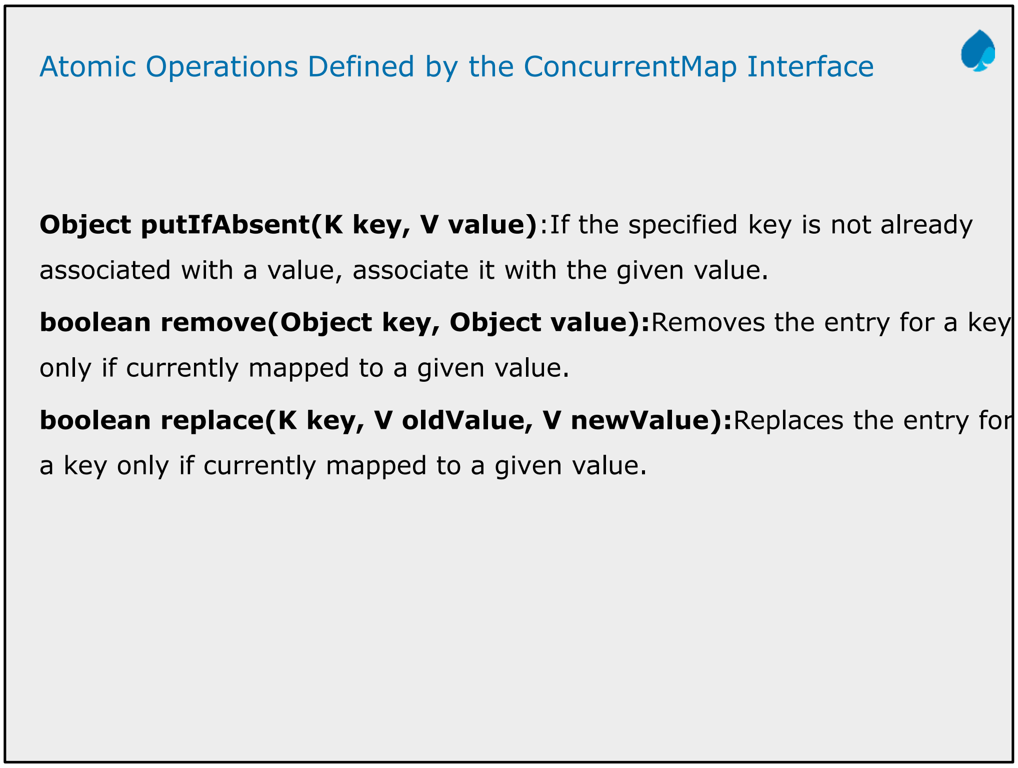
Introducing Concurrent Maps and Their Implementations

ConcurrentMap : ConcurrentMap is an interface, which is introduced in JDK 1.5 represents a Map which is capable of handling concurrent access to it and ConcurrentMap interface present in java.util.concurrent package.The ConcurrentMap provides some extra methods apart from what it inherits from the SuperInterface i.e. java.util.Map

ConcurrentHashMap : ConcurrentHashMap is enhancement of HashMap as we know that while dealing with Threads in our application HashMap is not a good choice because performance wise HashMap is not up to the mark.

ConcurrentSkipListMap : is an implementation of ConcurrentNavigableMap provided in the JDK since 1.6. The elements are sorted based on their natural sorting order of keys. The order can be customized using a Comparator provided during the time of initialization.

Key points of ConcurrentHashMap:
The underlined data structure for ConcurrentHashMap is Hashtable.
ConcurrentHashMap class is thread-safe i.e. multiple thread can operate on a single object without any complications.
At a time any number of threads are applicable for read operation without locking the ConcurrentHashMap object which is not there in HashMap.
In ConcurrentHashMap, the Object is divided into number of segments according to the concurrency level.
Default concurrency-level of ConcurrentHashMap is 16.
In ConcurrentHashMap, at a time any number of threads can perform retrieval operation but for updation in object, thread must lock the particular segment in which thread want to operate.This type of locking mechanism is known as **Segment locking or bucket locking**.Hence at a time 16 updation operations can be performed by threads.
null insertion is not possible in ConcurrentHashMap as key or value.



Atomic Operations Defined by the ConcurrentMap Interface

Object putIfAbsent(K key, V value):If the specified key is not already associated with a value, associate it with the given value.

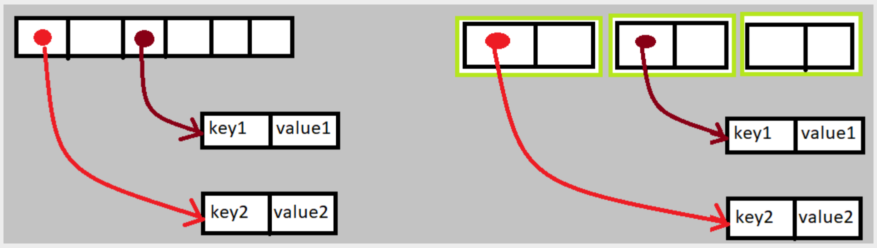
boolean remove(Object key, Object value): Removes the entry for a key only if currently mapped to a given value.

boolean replace(K key, V oldValue, V newValue): Replaces the entry for a key only if currently mapped to a given value.

Understanding the Structure of the ConcurrentHashMap from Java 7

It allows concurrent access to the map. Part of the map called Segment (internal data structure) is only getting locked while adding or updating the map. So ConcurrentHashMap allows concurrent threads to read the value without locking at all. This data structure was introduced to improve performance.

synchronizing array itself synchronizing parts of array



Introducing the Java 8 ConcurrentHashMap and Its Parallel Methods

Implementation is completely changed.

Serialization : compatible with jdk7 in both ways

Tailored to handle heavy concurrency and millions of key/values pairs

Parallel methods implemented.

Java 8 introduced the forEach, search, and reduce methods, which are pretty much to support parallelism. These three operations are available in four forms: accepting functions with keys, values, entries, and key-value pair arguments.

Parallel Search on a Java 8 ConcurrentHashMap

Consider Below code :

```
ConcurrentHashMap<String, String> hashMap = new
ConcurrentHashMap<>();
hashMap.put("A", “Pune”);
hashMap.put("B", “Mumbai”);
hashMap.put("C", “Patiyala”);
hashMap.put("D", “Panjim”);

String result= hashMap.search (10,
                              (key,value) -> value.startsWith(“P”)? “P” :null);
```

The first Parameter is parallelism threshold i.e. a number of key/value pairs that will trigger parallel search in this map.

The second is the operation to be applied

searchKeys(),searchValues(),searchEntries()

Parallel Map/Reduce on a Java 8 ConcurrentHashMap

Consider below Code :

```
ConcurrentHashMap<String, Integer> hashMap = new
ConcurrentHashMap<>();
hashMap.put("A", 1);
hashMap.put("B", 2);
hashMap.put("C", 3);
hashMap.put("D", 4);

String result= hashMap.reduce (10,
                                (key,value) -> value.size(),
                                  (value1,value2)->Integer.max(value1,value2));
```

The first bifunction maps to the element to be reduced
The second bifunction reduces 2 elements together

Parallel ForEach on a Java 8 ConcurrentHashMap

Consider Below Code :

```
ConcurrentHashMap<String, Integer> hashMap = new
ConcurrentHashMap<>();
hashMap.put("A", “Cpp”);
hashMap.put("B", “Java”);
hashMap.put("C", “Dotnet”);
hashMap.put("D", “Servlet”);


String result= hashMap.forEach(1000,
                          (key,value) -> value.removeIf(s->s.length() >=4));
```

The biconsumer function is applied to all the key/value pairs of the map

forEachKeys(),forEachValues() ,forEachEntry()

Creating a Concurrent Set on a Java 8 ConcurrentHashMap

```
Set<String> set= ConcurrentHashMap.newKeysSet();
```

No parallel operations from ConcurrentHashMap are available on this concurrentSet

Introducing Skip Lists to Implement ConcurrentMap

Another Concurrent Map (JKD 6)

A skip List is a smart structure used to create linked lists

Relies on atomic reference operations ,no synchronization

That can be used to create maps and sets

Understanding How Linked Lists Can Be Improved by Skip Lists

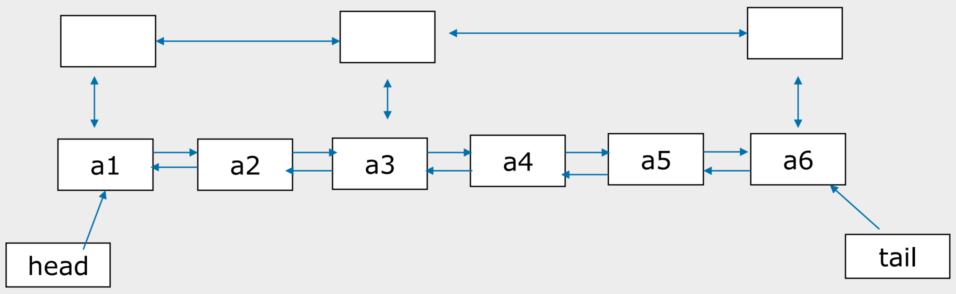
In case of linked lists to reach Nth element ,it takes O(N) time complexity.

If we double the linked list size ,the mean time taken to reach Nth element again doubles.

What is the solution on this ?

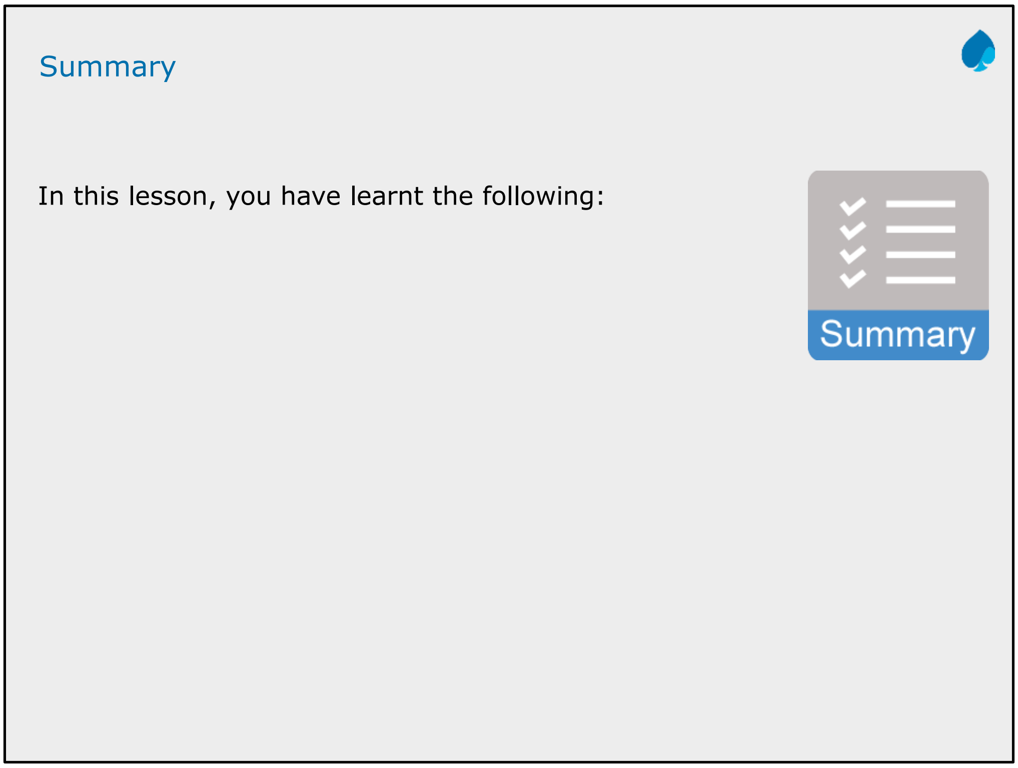
--Create a fast access list

The access time is now O(log(N))



Demo

Lab



Summary

In this lesson, you have learnt the following:

Add the notes here.

Review Question