Java Enterprise Edition

ANUDIP FOUNDATION

Concurrent Collections in java

Objective: • Collections • Mapping techniques • Hashmap • Error Handling	**Materials Required:** 1. Eclipse IDE/IntelliJ/STC
Theory:120mins	**Practical:120mins**
Total Duration: 240 mins	

· Concurrent Collections
o Implementing Concurrency at the API Level
o Hierarchy of Collection and Map, Concurrent Interfaces
o What Does It Mean for an Interface to Be Concurrent?
o Why You Should Avoid Vectors and Stacks
o Understanding Copy On Write Arrays
o Introducing Queue and Deque, and Their Implementations
o Understanding How Queue Works in a Concurrent Environment
o Adding Elements to a Queue That Is Full: How Can It Fail?
o Understanding Error Handling in Queue and Deque
o Introducing Concurrent Maps and Their Implementations
o Atomic Operations Defined by the ConcurrentMap Interface
o Understanding Concurrency for a HashMap
o Understanding the Structure of the ConcurrentHashMap from Java 7
o Introducing the Java 8 ConcurrentHashMap and Its Parallel Methods
o Parallel Search on a Java 8 ConcurrentHashMap
o Parallel Map / Reduce on a Java 8 ConcurrentHashMap
o Parallel ForEach on a Java 8 ConcurrentHashMap
o Creating a Concurrent Set on a Java 8 ConcurrentHashMap
o Introducing Skip Lists to Implement ConcurrentMap
o Understanding How Linked Lists Can Be Improved by Skip Lists
o How to Make a Skip List Concurrent Without Synchronization

Java concurrency API

The Java programming language has a very rich concurrency API. It contains classes to manage the basic elements of concurrency, such as Thread, Lock, and Semaphore, and classes that implement very high-level synchronization mechanisms, such as the **executor framework** or the new parallel Stream API.

In this section, we will cover the basic classes that form the concurrency API.

Basic concurrency classes

The basic classes of the Java concurrency API are:

- The Thread class: This class represents all the threads that execute a concurrent Java application
- The Runnable interface: This is another way to create concurrent applications in Java
- The ThreadLocal class: This is a class to store variables locally to a thread
- The ThreadFactory interface: This is the base of the Factory design pattern that you can use to create customized threads

Synchronization mechanisms

The Java concurrency API includes different synchronization mechanisms that allow you to:

- Define a critical section to access a shared resource
- Synchronize different tasks in a common point

The following mechanisms are considered to be the most important synchronization mechanisms:

- The synchronized keyword: The synchronized keyword allows you to define a critical section in a block of code or in an entire method.
- The Lock interface: Lock provides a more flexible synchronization operation than the synchronized keyword. There are different kinds of Locks: ReentrantLock, to implement a Lock that can be associated with a condition; ReentrantReadWriteLock, which separates read and write operations; and StampedLock, a new feature of Java 8 that includes three modes for controlling read/write access.
- The Semaphore class: The class that implements the classical semaphore to implement synchronization. Java supports binary and general semaphores.
- The CountDownLatch class: A class that allows a task to wait for the finalization of multiple operations.
- The CyclicBarrier class: A class that allows the synchronization of multiple threads in a common point.
- The Phaser class: A class that allows you to control the execution of tasks divided into phases. None of the tasks advance to the next phase until all of the tasks have finished the current phase.

Executors

The executor framework is a mechanism that allows you to separate thread creation and management for the implementation of concurrent tasks. You don't have to worry about the creation and management of threads, only about creating tasks and sending them to the executor. The main classes involved in this framework are:

- The Executor and ExecutorService interface: They include methods common to all executors.
- ThreadPoolExecutor: This is a class that allows you to get an executor with a pool of threads and optionally define a maximum number of parallel tasks
- ScheduledThreadPoolExecutor: This is a special kind of executor to allow you to execute tasks after a delay or periodically
- Executors: This is a class that facilitates the creation of executors

- The Callable interface: This is an alternative to the **Runnable** interface—a separate task that can return a value
- The Future interface: This is an interface that includes the methods to obtain the value returned by a Callable interface and to control its status

The Fork/Join framework

The **Fork/Join framework** defines a special kind of executor specialized in the resolution of problems with the divide and conquer technique. It includes a mechanism to optimize the execution of the concurrent tasks that solve these kinds of problems. Fork/Join is specially tailored for fine-grained parallelism as it has a very low overhead in order to place the new tasks into the queue and take queued tasks for execution. The main classes and interfaces involved in this framework are:

- ForkJoinPool: This is a class that implements the executor that is going to run the tasks
- ForkJoinTask: This is a task that can be executed in the ForkJoinPool class
- ForkJoinWorkerThread: This is a thread that is going to execute tasks in the ForkJoinPool class

Parallel streams

Streams and **Lambda expressions** are maybe the two most important new features of the Java 8 version. Streams have been added as a method in the Collection interface and other data sources and allow processing all elements of a data structure, generating new structures, filtering data and implementing algorithms using the map and reduce technique.

A special kind of stream is a parallel stream which realizes its operations in a parallel way. The most important elements involved in the use of parallel streams are:

- The Stream interface: This is an interface that defines all the operations that you can perform on a stream.

- Optional: This is a container object that may or may not contain a non-null value.
- Collectors: This is a class that implements reduction operations that can be used as part of a stream sequence of operations.
- Lambda expressions: Streams has been thought to work with Lambda expressions. Most stream methods accept a lambda expression as a parameter. This allows you to implement a more compact version of the operations.

Concurrent data structures

Normal data structures of the Java API (`ArrayList`, `Hashtable`, and so on) are not ready to work in a concurrent application unless you use an external synchronization mechanism. If you use it, you will be adding a lot of extra computing time to your application. If you don't use it, it's probable that you will have race conditions in your application. If you modify them from several threads and a race condition occurs, you may experience various exceptions thrown (such as, `ConcurrentModificationException` and `ArrayIndexOutOfBoundsException`), there may be silent data loss or your program may even stuck in an endless loop.

The Java concurrency API includes a lot of data structures that can be used in concurrent applications without risk. We can classify them in two groups:

- **Blocking data structures**: These include methods that block the calling task when, for example, the data structure is empty and you want to get a value.
- **Non-blocking data structures:** If the operation can be made immediately, it won't block the calling tasks. Otherwise, it returns the null value or throws an exception.

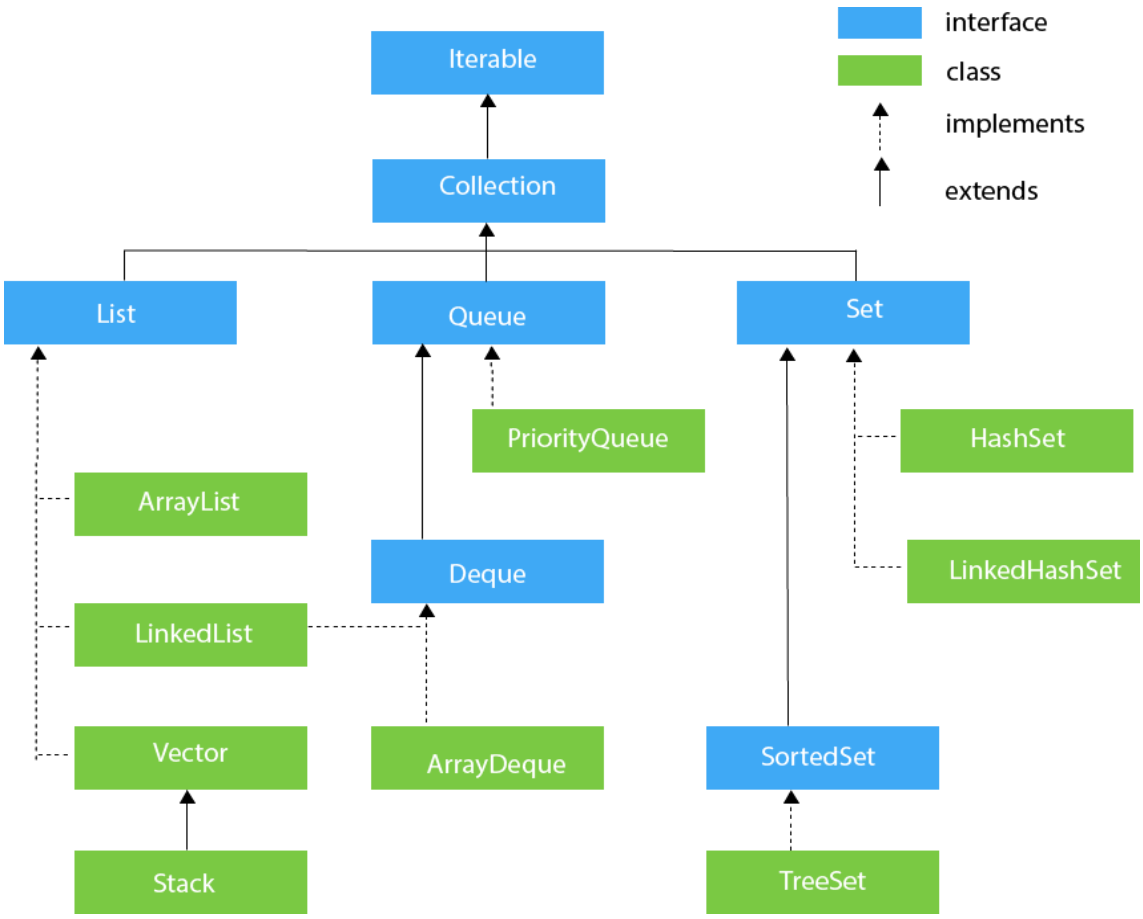
These are some of the data structures:

- ConcurrentLinkedDeque: This is a non-blocking list
- ConcurrentLinkedQueue: This is a non-blocking queue
- LinkedBlockingDeque: This is a blocking list

- LinkedBlockingQueue: This is a blocking queue
- PriorityBlockingQueue: This is a blocking queue that orders its elements based on its priority
- ConcurrentSkipListMap: This is a non-blocking navigable map
- ConcurrentHashMap: This is a non-blocking hash map
- AtomicBoolean, AtomicInteger, AtomicLong, and AtomicReference: These are atomic implementations of the basic Java data types

Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The **java.util** package contains all the classes and interfaces for the Collection framework.



Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

No.	Method	Description
1	public boolean add(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	default boolean removeIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.
6	public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.
7	public int size()	It returns the total number of elements in the collection.
8	public void clear()	It removes the total number of elements from the collection.
9	public boolean contains(Object element)	It is used to search an element.

10	public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
11	public Iterator iterator()	It returns an iterator.
12	public Object[] toArray()	It converts collection into array.
13	public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
14	public boolean isEmpty()	It checks if collection is empty.
15	default Stream<E> parallelStream()	It returns a possibly parallel Stream with the collection as its source.
16	default Stream<E> stream()	It returns a sequential Stream with the collection as its source.
17	default Spliterator<E> spliterator()	It generates a Spliterator over the specified elements in the collection.
18	public boolean equals(Object element)	It matches two collections.
19	public int hashCode()	It returns the hash code number of the collection.

Iterator interface

Iterator interface provides the facility of iterating the elements in a forward direction only.

Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is less used.

Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

1. Iterator<T> iterator()

It returns the iterator over the elements of type T.

Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

```
1. List <data-type> list1= new ArrayList();
2. List <data-type> list2 = new LinkedList();
3. List <data-type> list3 = new Vector();
4. List <data-type> list4 = new Stack();
```

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

The classes that implement the List interface are given below.

ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

```
1. import java.util.*;
2. class TestJavaCollection1{
3. public static void main(String args[]){
4. ArrayList<String> list=new ArrayList<String>();//Creating arraylist
5. list.add("Ravi");//Adding object in arraylist
6. list.add("Vijay");
7. list.add("Ravi");
8. list.add("Ajay");
9. //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. while(itr.hasNext()){
12. System.out.println(itr.next());
13. }
14. }
15. }
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection2{
3. public static void main(String args[]){
4. LinkedList<String> al=new LinkedList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. Iterator<String> itr=al.iterator();
10. while(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection3{
3. public static void main(String args[]){
4. Vector<String> v=new Vector<String>();
5. v.add("Ayush");
6. v.add("Amit");
7. v.add("Ashish");
8. v.add("Garima");
9. Iterator<String> itr=v.iterator();
10. while(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }
```

Output:

```
Ayush
Amit
Ashish
Garima
```

Stack

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection4{
3. public static void main(String args[]){
4. Stack<String> stack = new Stack<String>();
5. stack.push("Ayush");
6. stack.push("Garvit");
7. stack.push("Amit");
8. stack.push("Ashish");
```

```
9. stack.push("Garima");
10. stack.pop();
11. Iterator<String> itr=stack.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

Output:

```
Ayush
Garvit
Amit
Ashish
```

Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

```
1. Queue<String> q1 = new PriorityQueue();
2. Queue<String> q2 = new ArrayDeque();
```

There are various classes that implement the Queue interface, some of them are given below.

PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection5{
3. public static void main(String args[]){
```

```
4. PriorityQueue<String> queue=new PriorityQueue<String>();
5. queue.add("Amit Sharma");
6. queue.add("Vijay Raj");
7. queue.add("JaiShankar");
8. queue.add("Raj");
9. System.out.println("head:"+queue.element());
10. System.out.println("head:"+queue.peek());
11. System.out.println("iterating the queue elements:");
12. Iterator itr=queue.iterator();
13. while(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. queue.remove();
17. queue.poll();
18. System.out.println("after removing two elements:");
19. Iterator<String> itr2=queue.iterator();
20. while(itr2.hasNext()){
21. System.out.println(itr2.next());
22. }
23. }
24. }
```

Output:

```
head:Amit Sharma
head:Amit Sharma
iterating the queue elements:
Amit Sharma
Raj
JaiShankar
Vijay Raj
after removing two elements:
Raj
Vijay Raj
```

Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

```
1. Deque d = new ArrayDeque();
```

ArrayDeque

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection6{
3. public static void main(String[] args) {
4. //Creating Deque and adding elements
5. Deque<String> deque = new ArrayDeque<String>();
6. deque.add("Gautam");
7. deque.add("Karan");
8. deque.add("Ajay");
9. //Traversing elements
10. for (String str : deque) {
11. System.out.println(str);
12. }
13. }
14. }
```

Output:

```
Gautam
Karan
Ajay
```

Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

```
1. Set<data-type> s1 = new HashSet<data-type>();
2. Set<data-type> s2 = new LinkedHashSet<data-type>();
3. Set<data-type> s3 = new TreeSet<data-type>();
```

HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection7{
3. public static void main(String args[]){
4. //Creating HashSet and adding elements
5. HashSet<String> set=new HashSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=set.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

Output:

```
Vijay
Ravi
Ajay
```

LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection8{
3. public static void main(String args[]){
4. LinkedHashSet<String> set=new LinkedHashSet<String>();
5. set.add("Ravi");
6. set.add("Vijay");
7. set.add("Ravi");
8. set.add("Ajay");
9. Iterator<String> itr=set.iterator();
10. while(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }
```

Output:

```
Ravi
Vijay
Ajay
```

SortedSet Interface

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

```
1. SortedSet<data-type> set = new TreeSet();
```

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

```
1. import java.util.*;
2. public class TestJavaCollection9{
3. public static void main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> set=new TreeSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //traversing elements
11. Iterator<String> itr=set.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

Output:

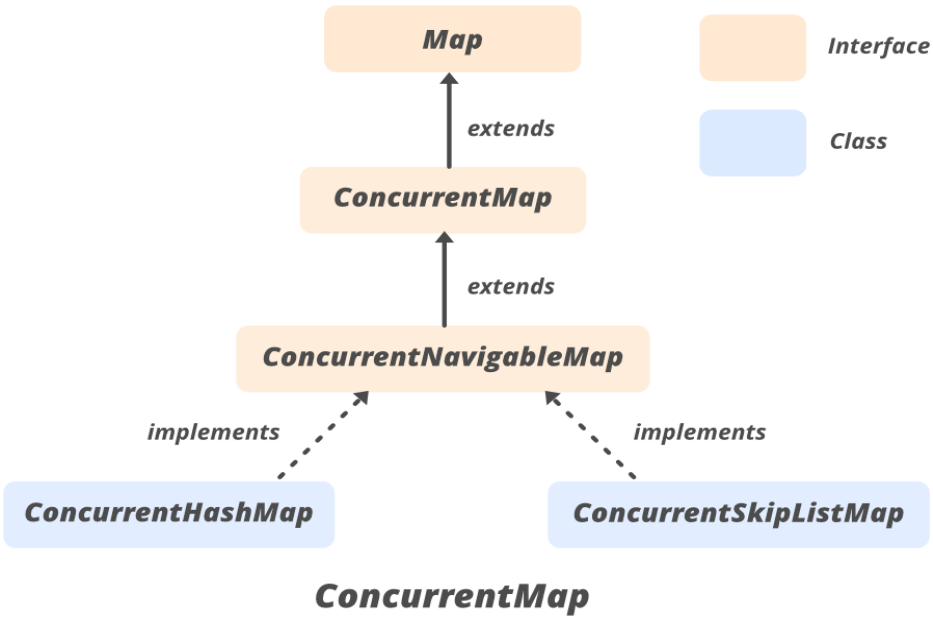
```
Ajay
Ravi
Vijay
```

ConcurrentMap Interface in java

ConcurrentMap is an interface and it is a member of the Java Collections Framework, which is introduced in JDK 1.5 represents a Map that is capable of handling concurrent access to it without affecting the consistency of entries in a map. ConcurrentMap interface present in **java.util.concurrent** package. It provides some extra methods apart from what it inherits from the SuperInterface i.e. java.util.Map. It has inherited the Nested Interface Map.Entry<K, V>.

HashMap operations are not synchronized, while Hashtable provides synchronization. Though Hashtable is a thread-safe, it is not very efficient. To solve this issue, the Java Collections Framework introduced **ConcurrentMap** in Java 1.5.

TheHierarchyofConcurrentMap



ConcurrentMap

Declaration:

public interface ConcurrentMap<K,V> extends Map<K,V>

Here, **K** is the type of key Object and **V** is the type of value Object.

- It extends the Map interface in Java.
- ConcurrentNavigableMap<K,V> is the SubInterface.
- ConcurrentMap is implemented by ConcurrentHashMap, **ConcurrentSkipListMap** classes.
- ConcurrentMap is known as a synchronized Map.

Implementing Classes

Since it belongs to **java.util.concurrent** package, we must import is using
import java.util.concurrent.ConcurrentMap

or

import java.util.concurrent.*

The ConcurrentMap has two implementing classes which are **ConcurrentSkipListMap** and **ConcurrentHashMap**. The ConcurrentSkipListMap is a scalable implementation of the ConcurrentNavigableMap interface which extends ConcurrentMap interface. The keys in ConcurrentSkipListMap are sorted by natural order or by using a Comparator at the time of construction of the object. The ConcurrentSkipListMap has the expected time cost of **log(n)** for insertion, deletion, and searching operations. It is a thread-safe class, therefore, all basic operations can be accomplished concurrently.

Syntax:

```
// ConcurrentMap implementation by ConcurrentHashMap
CocurrentMap<K, V> numbers = new ConcurrentHashMap<K, V>();

// ConcurrentMap implementation by ConcurrentSkipListMap
ConcurrentMap< ? , ? > objectName = new ConcurrentSkipListMap< ? , ? >();
```

Example:

- Java

```
// Java Program to illustrate methods
// of ConcurrentMap interface
import java.util.concurrent.*;

class ConcurrentMapDemo {

    public static void main(String[] args)
    {
        // Since ConcurrentMap is an interface,
        // we create instance using ConcurrentHashMap
        ConcurrentMap<Integer, String> m = new ConcurrentHashMap<Integer,
String>();
        m.put(100, "Geeks");
        m.put(101, "For");
```

```
        m.put(102, "Geeks");


        // Here we cant add Hello because 101 key
        // is already present
        m.putIfAbsent(101, "Hello");


        // We can remove entry because 101 key
        // is associated with For value
        m.remove(101, "For");


        // Now we can add Hello
        m.putIfAbsent(101, "Hello");


        // We can replace Hello with For
        m.replace(101, "Hello", "For");
        System.out.println("Map contents : " + m);
    }
}
```

Output
Map contents : {100=Geeks, 101=For, 102=Geeks}

Basic Methods

1. Add Elements

The put() method of ConcurrentSkipListMap is an in-built function in Java which associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced.

- Java

```
// Java Program to demonstrate adding

// elements



import java.util.concurrent.*;



class AddingElementsExample {

    public static void main(String[] args)

    {

        // Instantiate an object

        // Since ConcurrentMap

        // is an interface so We use

        // ConcurrentSkipListMap

        ConcurrentMap<Integer, Integer> mpp = new ConcurrentSkipListMap<Integer,
Integer>();
```

```
        // Adding elements to this map
          // using put() method
        for (int i = 1; i <= 5; i++)
            mpp.put(i, i);

        // Print map to the console
        System.out.println("After put(): " + mpp);
    }
}
```

Output
After put(): {1=1, 2=2, 3=3, 4=4, 5=5}

2. Remove Elements
The remove() method of ConcurrentSkipListMap is an in-built function in Java which removes the mapping for the specified key from this map. The method returns null if there is no mapping for that particular key. After this method is performed the size of the map is reduced.

- Java

```
// Java Program to demonstrate removing
// elements

import java.util.concurrent.*;
```

```
class RemovingElementsExample {

    public static void main(String[] args)

    {

        // Instantiate an object

        // Since ConcurrentMap

        // is an interface so We use

        // ConcurrentSkipListMap

        ConcurrentMap<Integer, Integer> mpp = new ConcurrentSkipListMap<Integer,
Integer>();



        // Adding elements to this map

        // using put method

        for (int i = 1; i <= 5; i++)

            mpp.put(i, i);



        // remove() mapping associated

        // with key 1

        mpp.remove(1);
```

```
        System.out.println("After remove(): " + mpp);
    }
}
```

Output
After remove(): {2=2, 3=3, 4=4, 5=5}

3. Accessing the Elements
We can access the elements of a ConcurrentSkipListMap using the get() method, the example of this is given below.

- Java

```
// Java Program to demonstrate accessing
// elements

import java.util.concurrent.*;

class AccessingElementsExample {

    public static void main(String[] args)
    {

        // Instantiate an object
```

```
        // Since ConcurrentMap

        // is an interface so We use

        // ConcurrentSkipListMap

        ConcurrentMap<Integer, String> chm = new ConcurrentSkipListMap<Integer,
String>();



        // insert mappings using put method

        chm.put(100, "Geeks");

        chm.put(101, "for");

        chm.put(102, "Geeks");

        chm.put(103, "Contribute");



        // Displaying the HashMap

        System.out.println("The Mappings are: ");

        System.out.println(chm);



        // Display the value of 100

        System.out.println("The Value associated to "

                        + "100 is : " + chm.get(100));
```

```
        // Getting the value of 103
        System.out.println("The Value associated to "
                           + "103 is : " + chm.get(103));
    }
}
```

Output
The Mappings are:

{100=Geeks, 101=for, 102=Geeks, 103=Contribute}

The Value associated to 100 is : Geeks

The Value associated to 103 is : Contribute

4. Traversing
We can use the Iterator interface to traverse over any structure of the Collection Framework. Since Iterators work with one type of data we use Entry< ? , ? > to resolve the two separate types into a compatible format. Then using the next() method we print the elements of the ConcurrentSkipListMap.

- Java

```
import java.util.concurrent.*;
import java.util.*;

public class TraversingExample {

    public static void main(String[] args)
    {
```

```
// Instantiate an object

// Since ConcurrentMap

// is an interface so We use

// ConcurrentSkipListMap

ConcurrentMap<Integer, String> chmap = new
ConcurrentSkipListMap<Integer, String>();



// Add elements using put()

chmap.put(8, "Third");

chmap.put(6, "Second");

chmap.put(3, "First");

chmap.put(11, "Fourth");



// Create an Iterator over the

// ConcurrentSkipListMap

Iterator<ConcurrentSkipListMap

        .Entry<Integer, String> > itr

   = chmap.entrySet().iterator();
```

```
        // The hasNext() method is used to check if there is
        // a next element The next() method is used to
        // retrieve the next element
        while (itr.hasNext()) {
            ConcurrentSkipListMap
                .Entry<Integer, String> entry
                = itr.next();
            System.out.println("Key = " + entry.getKey()
                               + ", Value = "
                               + entry.getValue());
        }
    }
}
```

Output

Key = 3, Value = First

Key = 6, Value = Second

Key = 8, Value = Third

Key = 11, Value = Fourth

Methods of ConcurrentMap

- **K** – The type of the keys in the map.
- **V** – The type of values mapped in the map.

Why You Should Avoid Vectors and Stacks

Isn't its use valid when working with concurrency?

And if I don't want to manually synchronize objects and just want to use a thread-safe collection without needing to make fresh copies of the underlying array (as CopyOnWriteArrayList does), then is it fine to use Vector?

What about Stack, which is a subclass of Vector, what should I use instead of it?
Vector synchronizes on each individual operation. That's almost never what you want to do.

Generally you want to synchronize a *whole sequence* of operations. Synchronizing individual operations is both less safe (if you iterate over a Vector, for instance, you still need to take out a lock to avoid anyone else changing the collection at the same time, which would cause a ConcurrentModificationException in the iterating thread) but also slower (why take out a lock repeatedly when once will be enough)?

Of course, it also has the overhead of locking even when you don't need to.

Basically, it's a very flawed approach to synchronization in most situations. you can decorate a collection using the calls such as Collections.synchronizedList - the fact that Vector combines both the "resized array" collection implementation with the "synchronize every operation" bit is another example of poor design; the decoration approach gives cleaner separation of concerns.

As for a Stack equivalent - I'd look at Deque/ArrayDeque to start with.

How to Make a Skip List Concurrent Without Synchronization

Most of the classes in Java's Collections Framework are unsynchronized by default, but can be made into something synchronized if you need them to be thread-safe. The synchronization has a performance penalty, so if you're writing something that doesn't need to be thread-safe, you're better off with the unsynchronized version.

But ConcurrentSkipListMap doesn't follow this scheme. There is no unsynchronized version. Why is there not a faster unsynchronized SkipListMap for applications that don't require thread safety, in line with the rest of the Collections Framework?

All I can think is that the simplest implementation of a Skip List is already thread-safe, so there would be no performance penalty for having a synchronized version. This would make some kind of sense, but a look at the source code doesn't quite bear that out. Although there are no synchronized blocks in the code, the Javadoc does start with

This class implements a concurrent variant of SkipLists...

which suggests that it's going out of its way to modify the algorithm to make it thread-safe. Later on, we read

The basic idea in these lists is to mark the "next" pointers of deleted nodes when deleting to avoid conflicts with concurrent insertions...

which also sounds as though there is some kind of overhead involved.

Is it just that this overhead is so small that it's not worth having a non-thread-safe SkipListMap?

Although Java doesn't have a SkipListMap, it has a non-synchronized counterpart for ConcurrentSkipListMap - TreeMap

Both TreeMap and ConcurrentSkipListMap implementes SortedMap, NavigableMap and are sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

But TreeMap uses Red-Black tree internally and ConcurrentSkipListMap uses concurrent variant of SkipLists as mentioned in java docs.

So, if you want a unsynchronized version of ConcurrentSkipListMap, you can use TreeMap