### Collection

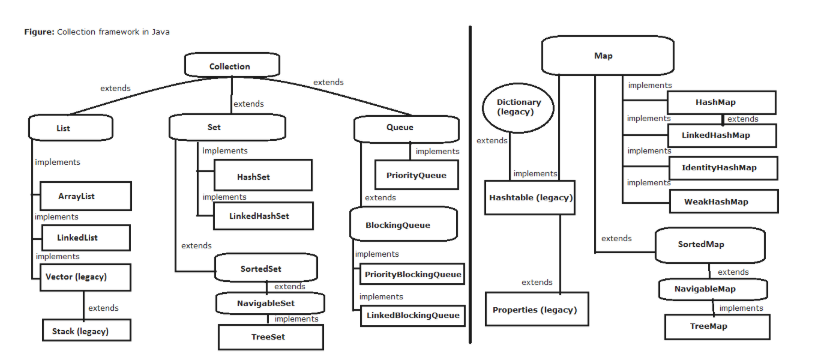
**Collection** is group of element/objects represented as single unit/entity.

Programmers can perform various operations like insertion, deletion, sorting, searching and reversing on the collection.

A Collection framework consists of various classes & interfaces for different operational purposes.

**A Collection framework are used to store group of objects as single unit/entity**.

It's dynamically growable in nature.



**ArrayList**

- ArrayList uses resizable array or growable array to store the elements/objects.

- Duplicate elements are allowed to be inserted.

- Allows null insertion (no limit, we can insert any number of NULL to ArrayList)

- ArrayList is non-synchronized

- Insertion order is not maintained.

- **Elements retrieval is fast as ArrayList works on the zero-based index**.

- Manipulation (addition, deletion of element from middle of ArrayList) is very slow as **it requires lot of shifting work internally**.

- it is **very safe to remove element from ArrayList while iterating using iterator**.

- Compiler does not throw any **runtime exception** like **UnSupportedOperationException**.

**LinkedList**

- LinkedList uses doubly linked list to store the elements/objects.

- Duplicate elements are allowed to be inserted.

- Allows null insertion (no limit, we can insert any number of NULL to linkedList)

- LinkedList is non-synchronized

- Insertion order is maintained.

- **Elements retrieval is much slower, as LinkedList need to iterate over complete list to retrieve or get the required element.**

- **Manipulation** (addition, deletion of element from middle of LinkedList) is **very fast, as it works on basis of previous and next element. There is no shifting is required.**

addFirst() getFirst() removeFirst()

addLast() getLast() removeLast()

**Vector**

- Vector is **legacy class**.

- Vector uses resizable array or growable array to store the elements/objects.

- All its methods are synchronized i.e. thread safe for any operation.

- Vector is exactly same as that of ArrayList except all the methods of the vector class are **synchronized** (i.e.) thread safe.

- Duplicate elements are allowed to be inserted.

- Allows null insertion (no limit, we can insert any number of NULL to vector)

- Vector is synchronized

- **Insertion order is maintained**.

addElement() removeElement() firstElement() lastElement()

**Stack**

- Stack is legacy class

- works in Last In First Out (**LIFO**).

- Stack is sub class of vector.

- All properties are same as that of Vector class.

- Its data structure is designed in such a way that elements added last to the stack will be returned first.

- push () - add/insert the new element/object into stack.

- pop () - remove and returns top of the stack

- peek () - return top if the stack

**Insert at First, Remove at First** - **Stack** - LIFO

**Insert at Last, Remove at First** - **Queue** - FIFO

**CopyOnwriteArrayList**

- CopyOnwriteArrayList is **the thread safe version of ArrayList** where **modify operation is performed on separate cloned copy** and **finally the JVM merges both original and cloned copies.**

- For every update operation, a new separate cloned copy is created, and modification is performed on cloned copy, while other threads can iterate over original copy.

- After modification/updation, JVM takes care of merging both original and cloned copies so that we get latest copy with all modification/updation.

- **While one thread iterating over the original copy and other threads can modify with separate cloned copy and compiler won't throw any ConcurrentModificationException which is not in the case with ArrayList**.

- It never throws ConcurrentModificationException while 2 or more threads operating simultaneously. i.e. fail-safe iteration.

- But there are certain limitation too with CopyOnwriteArrayList which isn't case with ArrayList like, while iterating CopyOnwriteArrayList, remove operation is not a possible, and compiler throws **UnsupportedOperationException**.

- Duplicate elements are allowed to be inserted.

- Allows null insertion

- **CopyOnwriteArrayList is synchronized**

- **Insertion order is maintained.**

- present in java.util.concurrent package.

- addIfAbsent(Object ) to add new object to COWAL, if specified object isn't present in the invoking list.

**Set Interface**

- Set is sub interface of Collection interface set extends collection.

- To represent a group of element/objects as a single unit/entity, where duplicates aren't allowed.

- Set allows only unique elements/objects to be inserted.

- Set doesn't maintain insertion order.

- While iterating through set, we will get items in random order.

**HashSet**

- HashSet uses hashtable to store the elements/objects.

- Duplicate elements are not allowed. **If duplicate value is added again, then there won't be any compile time or run time errors. Simply add objects method returns false for already containing the object inside the HashSet.**

- HashSet **contains unique elements**.

- **Allows Null insertion but maximum of only one null value**.

- **HashSet is non-synchronized**.

- **While iterating through HashSet, we will get items in random order as against insertion order.**

- **Insertion order is not maintained.**

- **Search operation is faster i.e. searching any element from HashSet is faster, as it uses hashing to store the elements.**

**LinkedHashSet**

- LinkedHashSet is exactly same as that of HashSet but it preserves insertion order.

- LinkedHashSet uses the combination of LinkedList and HashTable to store the elements.

- While iterating through LinkedHashSet, we will get items as per insertion order.

- Insertion order is maintained.

**TreeSet**

- TreeSet **uses balanced tree** to store the elements/objects in sorting order.

- Insertion order is not maintained. It maintains ascending sorting order by default.

- From Java 1.7 version, Null insertion is not possible.

**HashMap**

- HashMap uses hash table data structure to store key-value pairs.

- HashMap **allows null insertion for key but maximum of only one. It allows only unique key**.

- **HashMap is non-synchronized in nature** i.e. HashMap is not thread safe.

- HashMap **store entries in random order**.

- While iterating through HashMap, we will get map entries in random order as against insertion order.

- **Insertion order is not maintained.**

- HashMap iterator is fail fast. It throws ConcurrentModificationException if there is structural modification of the collection.

- HashMap performance is relatively high because it is non-synchronized in nature and any number of threads can perform simultaneously.

**LinkedHashMap**

- LinkedHashMap is exactly same as that of HashMap but preserves insertion order.

- LinkedHashMap uses combination of LinkedList & HashTable to store map entries.

- LinkedHashMap allows only unique keys.

- LinkedHashMap allows null insertion for key but maximum of only one. It allows only unique key.

- **Insertion order is maintained.**

**- While iterating through LinkedHashMap, we will get map entries in insertion order.**

- LinkedHashMap is non-synchronized.

LinkedHashMap<String,Integer> lhm= **new** LinkedHashMap<>();

lhm.put("Sunday", 1);

lhm.put("Monday",2 );

lhm.put("Tuesday", 3);

lhm.put("Wednesday", 4);

lhm.putIfAbsent("Thrusday", 5);

**if**(!lhm.containsKey("Friday"))

lhm.put("Friday", 6);

**if**(!lhm.containsValue(7))

lhm.put("Saturday",7 );

lhm.put(**null**, 9);

lhm.putIfAbsent(**null**, 8);

**if**(!lhm.isEmpty()) {

System.***out***.println("Linked Hash map shows the data insertion order");

System.***out***.println(lhm);

}

**Output:**

Linked Hash map shows the data insertion order

{Sunday=1, Monday=2, Tuesday=3, Wednesday=4, Thrusday=5, Friday=6, Saturday=7, null=9}

**TreeMap**

- TreeMap uses **red-black-tree** to store key-value pairs in sorting order on the basis of keys only, not values.

- Null insertion is not possible in Java 1.7 version.

- TreeMap is non-synchronized.

- Insertion order is not maintained. Maintains ascending order by default.

TreeMap<String,Integer> tm= **new** TreeMap<>();

//tm.put(null, 8); //NullPointerException

tm.put("Sunday", 1);

tm.put("Monday",2 );

tm.put("Tuesday", 3);

tm.put("Wednesday", 4);

tm.putIfAbsent("Thrusday", 5);

**if**(!tm.containsKey("Friday"))

tm.put("Friday", 6);

**if**(!tm.containsValue(7))

tm.put("Saturday",7 );

**if**(!tm.isEmpty()) {

System.***out***.println("Tree map shows the data insertion order");

System.***out***.println(tm);

}

System.***out***.println("Tree Map Size "+tm.size());

System.***out***.println("Tree Map get 4 :"+tm.get("Wednesday"));

System.***out***.println("Tree Map get 8 :"+tm.get("July"));

**Output:**

Tree map shows the data insertion order

{Friday=6, Monday=2, Saturday=7, Sunday=1, Thrusday=5, Tuesday=3, Wednesday=4}

Tree Map Size 7

Tree Map get 4 :4

Tree Map get 8 :null

**IdentityHashMap**

- in HashMap, **JVM** uses **equals() method to check the uniqueness of keys before storing.**

- In IdentityHashMap, JVM uses == operator to check the uniqueness of keys before storing.

IdentityHashMap<Integer,String> ihm= **new** IdentityHashMap();

ihm.put(**new** Integer(1), "Guru");

ihm.put(**new** Integer(1), "Mangai");

ihm.put(**new** Integer(1), "Kumaresh");

ihm.put(**new** Integer(1), "Nikhilan");

ihm.put(1, "Guru Murugesan");

System.***out***.println(ihm);

System.***out***.println("Size:"+ihm.size());

System.***out***.println("ihm.get(1)"+ihm.get(1));

**Output:**

{1=Guru, 1=Kumaresh, 1=Guru Murugesan, 1=Nikhilan, 1=Mangai}

Size:5

ihm.get(1)Guru Murugesan

**WeakHashMap**

- WeakHashMap is exactly same as that of HashMap with few differences.

- An entry in the WeakHashMap will automatically removed by garbage collector when its key is no longer in ordinary use.

- **When the key has been discarded its entry is removed from the map in Java.**

WeakHashMap<Integer,String> whm = new WeakHashMap<>();

Integer i10 = new Integer (10);

Integer i5 = new Integer (5);

whm.put(i10, "Ten");

whm.put(i5, "Five");

System.out.println("Weak Hash MAp"+whm);

i10 = null;

i5=null;

System.gc();

System.out.println("After GC Weak Hash MAp"+whm);

**output**:

Weak Hash MAp{10=Ten, 5=Five}

After GC Weak Hash MAp{}

**HashTable**

- HashTable is used to store key-value pairs.

- HashTable does not allow null insertion for keys and values.

- HashTable is synchronized, all methods of HashTable is thread safe. (Only one thread is allowed to access at any given point of time)

- insertion order is not maintained.

- Search operation is faster i.e. searching any element from HashTable is faster, as it uses hashing to store key-value pairs.

**ConcurrentHashMap**

- ConcurrentHashMap is used to store a group of key-value pairs as a single unit/entity.

- ConcurrentHashMap uses hash table data structure to store key-value pairs (which is known as Map.Entry)

- **It is a thread safe version of HashMap**.

- **It allows concurrent access to read/modify Map Object**. one thread iterating never stops other thread to modify.

- It can be accessed by multiple threads and performs operations simultaneously.

- ConcurrentHashMap is fail safe and it will never throw ConcurrentModificationException while two or more threads operating simultaneously.

- Null insertions are not allowed for both key and values. It will throw NullPointerException.

- ConcurrentHashMap is also like HashMap how it differs is locking strategy used by the ConcurrentHashMap

- For Read operation, lock isn't required.

- But for Update operation, lock is required but it is only part of the map object.

- When updation is in process, thread locks the particular segment and it does not allow any other thread to perform updating (Put/Remove) in the same segment until lock is released on the segment.

- When thread locks one segment for updation (Put/remove), it doesn't block it for retrieval(Get). Hence some other thread can read the same segment, but it will be able to read the data before locking.

- Two threads can read data from same/different segment of ConcurrentHashMap at the same time without blocking each other.

- Two threads can write data on different segment at the same time.

- But two threads can't write data on the same segments at the same time. One has to wait for other to complete the operation.

Unlike HashTable, it doesn't synchronize every method. It is thread safe without synchronizing the whole map. The locking is at much granularity level.

**Different ways of Iteration of Maps**

Map<Integer, String> hm = **new** HashMap<>();

hm.put(1, "guru");

hm.put(2, "kumaresh");

hm.put(3, "nikhilan");

Iterator<Integer> itr = hm.keySet().iterator();

**while**(itr.hasNext()) {

**int** itrvalue = (Integer)itr.next();

System.***out***.println("Key"+itrvalue+" value:"+hm.get(itrvalue));

}

Iterator<String> iv = hm.values().iterator();

**while**(iv.hasNext()){

String values = (String)iv.next();

System.***out***.println("values"+values);

}

Iterator<Entry<Integer,String>> es = hm.entrySet().iterator();

**while**(es.hasNext()) {

Map.Entry me = (Map.Entry)es.next();

System.***out***.println("Key"+me.getKey()+ " value:"+me.getValue());

}

Set<Entry<Integer,String>> en = hm.entrySet();

**for**(Entry<Integer, String> s :en) {

System.***out***.println("key"+s.getKey() + " value"+s.getValue());

}

**for**(Map.Entry<Integer, String> me:hm.entrySet()) {

System.***out***.println("key"+me.getKey() + " value"+me.getValue());

}

System.***out***.println("\n"+"KeySet for Loop");

**for**(String key :hm.keySet()) {

System.***out***.println("key"+key +" Value:"+hm.get(key));

}

System.***out***.println("ForEach(k,v) Loop");

hm.forEach((k,v)-> {

System.***out***.println("Key:"+k+ " Value:"+v);

});

System.***out***.println("ForEach(k,v) Loop");

hm.entrySet().forEach((e)-> {

System.***out***.println("Key:"+e.getKey()+ " Value:"+e.getValue());

});

System.***out***.println("\n"+"Stream forEach(e) loop");

hm.entrySet().stream().forEach(e -> {

System.***out***.println("Key:"+e.getKey()+ " Value:"+e.getValue());

});

**Different ways of Iteration of Lists**

List<String> al = **new** ArrayList<>();

al.add("Apple");

al.add("Lemon");

al.add("Bannana");

al.add("Orange");

al.add("Musambi");

System.***out***.println("Print List "+al);

al.set(2, "Lemondate");

System.***out***.println("set 2 lemondate "+al);

al.add(4, "papaya");

System.***out***.println("Add 4 Papaya "+al);

System.***out***.println("get 4 "+al.get(4));

al.remove(1);

System.***out***.println("Remove index 1 "+al);

al.remove("papaya");

System.***out***.println("Remove Papaya "+al);

System.***out***.println("index of apple "+al.indexOf("Apple"));

System.***out***.println("last index of apple "+al.lastIndexOf("Apple"));

System.***out***.println("contains of apple "+al.contains("Apple"));

al.forEach(l->System.***out***.print(l+" "));

System.***out***.println("");

**for**(String l:al) {

System.***out***.print(l+" ");

}

System.***out***.println("");

Iterator<String> itr = al.iterator();

**while**(itr.hasNext()){

System.***out***.print(itr.next()+" ");

itr.remove();

}

**OUTPUT:**

Print List [Apple, Lemon, Bannana, Orange, Musambi]

set 2 lemondate [Apple, Lemon, Lemondate, Orange, Musambi]

Add 4 Papaya [Apple, Lemon, Lemondate, Orange, papaya, Musambi]

get 4 papaya

Remove index 1 [Apple, Lemondate, Orange, papaya, Musambi]

Remove Papaya [Apple, Lemondate, Orange, Musambi]

index of apple 0

last index of apple 0

contains of apple true

Apple Lemondate Orange Musambi

Apple Lemondate Orange Musambi

Apple Lemondate Orange Musambi

**java.util.Collections** is a utility class for operating on collection items.

**Collections read only**

Collections.unmodifiableList(list);

Collections.unmodifiableSet(set);

Collections.unmodifiableMap(map);

**How can we make synchronized**

List list = Collections.synchronizedList(list);

Set set = Collections.synchronizedSet(set);

Map map = Collections.synchronizedMap(map);

Only one thread is allowed to operate on synchronized list, by locking over complete list object.

Collections.reverse(al);

This method is used to reverse the order of array list contents.

Collections.sort(al);

To Sort arrayList contents in the ascending order.

Collections.sort(al, Collections.reverseOrder());

To reverse sort arrayList contents.

**Concurrent Modification**

When one or more threads is iterating over collection, in between, one thread changes the structure of the collection(either adding the element to the collection or by deleting the element in the collection or by updating the value at particular position in the collection) is known as Concurrent Modification.

**FailFast Iterator**

While iterating through the collection, instantly throws ConcurrentModificationException if there is structural modification of the collection.

FailFast Iterator can throw ConcurrentModificationException in 2 Scenarios.

**1. Single Threaded Environment**

- After the creation of iterator, structure is modified at any time by any method other than iterators.remove() method.

- Iterator reads internal data structure (Object array) directly.

- The Internal Data structure (Object array) should not be modified while iterating through collection.

- To ensure that it maintains the initial flag "Mods", Iterator checks the "Mod" flag whenever it gets the next() value using hasNext() and next() method.

- Value of mods flag changes whenever there is any structural modification, thus indicating iterator to throws ConcurrentModificationException.

**2. Multiple Threaded Environment**

- If one thread is modifying the structure of the collection while other threads is iterating over it.

**FailSafe Iterator**

- FailSafe Iterator makes a copy of internal data structure (Object Array) and iterate over the copied data structure.

- Any structural modification done to the iterator affects the copied data structure.

- So original data structure remains structurally unchanged.

- Hence no ConcurrentModificationException throws by the failsafe iterator.

**Two issues associated with FailSafe Iterators are**

1. Overhead of maintaining the copied data structure

2. FailSafe iterator does not guarantee that the data being read in the data currently in the original data structure.

|  |  |
| --- | --- |
| ArrayList | Dynamic Arrays |
| LinkedList | Doubly Linked List |
|  |  |
|  |  |
| HashSet | HashTable |
| LinkedHashSet | HashTable + Doubly LinkedList |
| TreeSet | Balanced Tree |
|  |  |
| HashMap | HashTable |
| LinkedHashMap | HashTable + Doubly LinkedList |
| TreeMap | RedBlackTree |
|  |  |

**ConcurrentModificationException in ArrayList**

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**import** java.util.ListIterator;

**public** **class** ConcurrentModificationExceptionALExample {

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

List<String> myList = **new** ArrayList<String>();

myList.add("1");

myList.add("2");

myList.add("3");

myList.add("4");

myList.add("5");

Iterator<String> it = myList.iterator();

**while** (it.hasNext()) {

String value = it.next();

System.***out***.println("Iterator Value:" + value);

**if** (value.equals("3")) {

// myList.remove(value); // Java.util.ConcurrentModificationException

it.remove(); //It allows remove operation through iterator

}

}

System.***out***.println("Iterator List :" +myList);

ListIterator<String> lit = myList.listIterator();

**while** (lit.hasNext()) {

String value = lit.next();

System.***out***.println("List Iterator Value:" + value);

// if (value.equals("4")) {

// myList.remove(value);

// }

**if** (value.equals("2")) {

lit.remove();

}

}

System.***out***.println("List Iterator List :" +myList);

}

}

**Output:**

Iterator Value:1

Iterator Value:2

Iterator Value:3

Iterator Value:4

Iterator Value:5

Iterator List :[1, 2, 4, 5]

List Iterator Value:1

List Iterator Value:2

List Iterator Value:4

List Iterator Value:5

List Iterator List :[1, 4, 5]

**ConcurrentModificationException in CopyOnWriteArrayList**

**import** java.util.Iterator;

**import** java.util.List;

**import** java.util.ListIterator;

**import** java.util.concurrent.CopyOnWriteArrayList;

**public** **class** ConcurrentModificationExceptionCOWALExample {

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

List<String> myList = **new** CopyOnWriteArrayList<String>();

myList.add("1");

myList.add("2");

myList.add("3");

myList.add("4");

myList.add("5");

Iterator<String> it = myList.iterator();

**while** (it.hasNext()) {

String value = it.next();

System.***out***.println("Iterator Value:" + value);

**if** (value.equals("3")) {

myList.remove(value); // COWAL allows remove operation in the list and not on the iteration

// it.remove(); //java.lang.UnsupportedOperationException

}

}

System.***out***.println("Iterator List :" +myList);

ListIterator<String> lit = myList.listIterator();

**while** (lit.hasNext()) {

String value = lit.next();

System.***out***.println("List Iterator Value:" + value);

**if** (value.equals("4")) {

myList.remove(value);

}

**if** (value.equals("2")) {

//lit.remove();

}

}

System.***out***.println("List Iterator List :" +myList);

}

}

**OUTPUT:**

Iterator Value:1

Iterator Value:2

Iterator Value:3

Iterator Value:4

Iterator Value:5

Iterator List :[1, 2, 4, 5]

List Iterator Value:1

List Iterator Value:2

List Iterator Value:4

List Iterator Value:5

List Iterator List :[1, 2, 5]

**ConcurrentModificationException in HashMap**

**import** java.util.HashMap;

**import** java.util.Iterator;

**import** java.util.Map;

**public** **class** ConcurrentModificationExceptionHMExample {

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

Map<String, String> myMap = **new** HashMap<String, String>();

myMap.put("1", "1");

myMap.put("2", "2");

myMap.put("3", "3");

Iterator<String> it1 = myMap.keySet().iterator();

**while** (it1.hasNext()) {

String key = it1.next();

System.***out***.println("Map Value:" + myMap.get(key));

**if** (key.equals("2")) {

myMap.put("1", "4"); // for already existing key, it doesn't throw any exception

//myMap.put("4", "4"); //Concurrent modificationException occurs while adding the data.

it1.remove(); // allows remove through iteration

//myMap.remove("3"); // ConcurrentModificationException occurs while removing data

}

}

System.***out***.println("Map Iterations List :" +myMap);

}

}

**OUTPUT:**

Map Value:1

Map Value:2

Map Value:3

Map Iterations List :{1=4, 3=3}

**ConcurrentModificationException in ConcurrentHashMap**

**import** java.util.Iterator;

**import** java.util.Map;

**import** java.util.concurrent.ConcurrentHashMap;

**public** **class** ConcurrentModificationExceptionCHMExample {

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

Map<String, String> myMap = **new** ConcurrentHashMap<String, String>();

myMap.put("1", "1");

myMap.put("2", "2");

myMap.put("3", "3");

Iterator<String> it1 = myMap.keySet().iterator();

**while** (it1.hasNext()) {

String key = it1.next();

System.***out***.println("Map Value:" + myMap.get(key));

**if** (key.equals("2")) {

myMap.put("1", "4"); // for already existing key, it doesn't throw any exception

myMap.put("4", "4"); //Concurrent modificationException will not occur while adding the data.

it1.remove(); // allows remove through iteration

myMap.remove("3"); // ConcurrentModificationException will not occurs while removing data

}

}

System.***out***.println("Map Iterations List :" +myMap);

}

}

**Output:**

Map Value:1

Map Value:2

Map Value:null

Map Value:4

Map Iterations List :{1=4, 4=4}