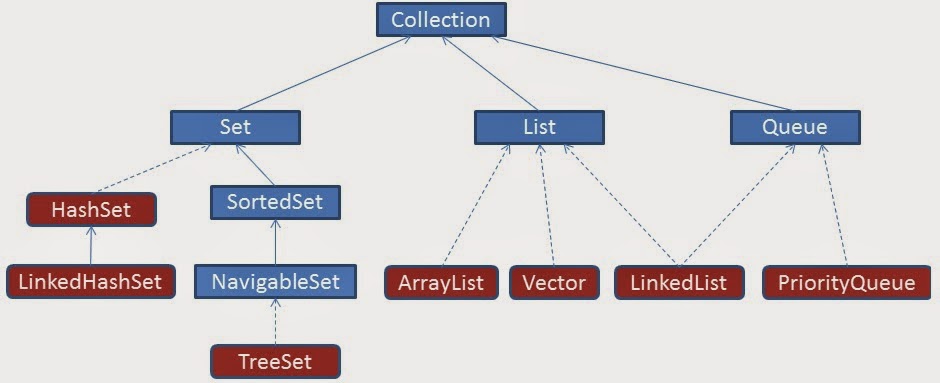
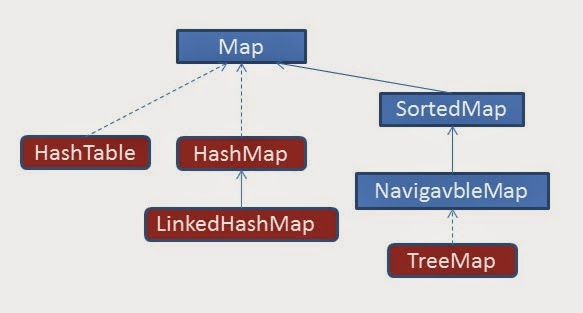
**Collections:**

[](http://2.bp.blogspot.com/-M0M8nv5s2lQ/U3BcbRQcRvI/AAAAAAAAAec/oBBmQCPDm9Y/s1600/Collection-Classes.tif)

[](http://4.bp.blogspot.com/-o9Jk4Z4Tohs/U3Be46CxGTI/AAAAAAAAAeo/Wq8-hhZ8dCA/s1600/Collection-Classes_Map.tif)

**\*What are the collections that you have worked on?**

1. List

2. Map

3. Set

4. Queue

5. Dequeue

**\*Map vs set?**

* **Set** contains values only whereas **Map** contains key and values both.
* **Set** contains unique values whereas **Map** can contain unique Keys with duplicate values.
* **Set** holds a single number of null value whereas **Map** can include a single null key with n number of null values.

**\*Linked List vs array list?**

**ArrayList** uses a dynamic array. ArrayList is better to store and fetch data. ArrayList provides random access. ArrayList takes less memory overhead as it stores only object.

**LinkedList** uses a doubly linked list. LinkedList is better to manipulate data. LinkedList does not provide random access. LinkedList takes more memory overhead, as it stores the object as well as the address of that object.

**\*Concurrent hashmap vs synchronized hashmap?**

| **Sr. No.** | **Key** | **Concurrent hash map** | **Synchronized hashmap** |
| --- | --- | --- | --- |
| 1 | Implementation | It is a class that implements a Concurrent hash map and serializable interface. | It is a method in Collection class. |
| 2 | Lock mechanism | Locks the portion | Locks the whole map. |
| 3 | Performance | Concurrent hashmap allows concurrent read and write. So performance is relatively better than a synchronized map. | Multiple threads can't access the map concurrently. So, performance is relatively less than a concurrent hash map. |
| 4 | Null key | It doesn't allow null as a key or value. | It allows null as a key. |
| 5 | Concurrent modification exception | It doesn't throw concurrent modification exception. | Iterator return by synchronized map throws concurrent modification exception |

**\*Intenal working of hashmap, hashset and arraylist?**

### 1. HashMap works on the principal of hashing. It stores values in the form of key-value pair and to access a value you need to provide the key. HashMap is basically a 2 dimensional Singly Linked List. It can grow in both directions.

### [Hash Map](http://4.bp.blogspot.com/-xGCOSWY473o/VZVKQ7kRlDI/AAAAAAAAAlw/27ajIG9FGf0/s1600/Presentation1.jpg)

### HashMap the 'key' element should implement equals() and hashcode() method. equals() method define that two objects are meaningfully equal. hashcode() helps HashMap to arrange elements separately in a bucket. So elements with same hascode are kept in the same bucket together.

### So when we want to fetch a element using get(K key), HashMap first identifies the bucket in which all elements of the same hascode as the hashcode of the 'key' passed are present. Now since it knows the bucket, it will only have to traverse through that bucket to fetch the actual object.

### For fast access to a value HashMap places an element (both key and value) in a [SinglyLinkedList](http://www.java-redefined.com/2013/08/data-structure-singly-linked-list.html)(Bucket). All the elements that have the same hascode are placed in the same SinglyLinkedList. The number of SinglyLinkedList(buckets) depends upon how many objects are present with different hashcode. To hold these buckets together a array is used. The size of the array is initially defined to 12. And it changes as new elements with different hascodes are added.

### HashMap also has some more variables which define the initial size of the array. DEFAULT\_LOAD\_FACTOR = 0.75f; DEFAULT\_INITIAL\_CAPACITY = 16;

### 2. Hashset is Non Duplicate and Un-Ordered.

**Hashset** is a special case.it is internally uses HashMap.HashSet has a instance variable called 'map' which is an instance of HashMap.

**add(E element)**  
When we add a value in Hashset, Hashset internally adds a value in 'map' by calling put(E,o);  
where E that is the key is the element passed in add(E element) of HashSet and 'o' as the value which is a dummy Object creted by doing *Object o = new Object;* which is common for all key's entered in HashMap 'map'.  
HashSet internally checks wether the Key that is 'element' is already present by calling the equals method of 'element'.  
This method returns false if the Key is already present in HashMap.

**3. ArrayList(Ordered)** works on the principle of creating a array and adding elements to it. So basically it internally just creates and array and adds elements to it. array has to be defined during its creation.

Integer [] a = new Integer[5];

Then how come ArrayList is dynamic? We will see this below. ArrayList class has a member variable elementData which is an Object array;  
**Object[] elementData;**  
When we do this: List l = new ArrayList();

The array **elementData** is initialised with a size of **10**.There are 2 important method in ArrayList:  
**1. add(E element)**  
When a new element is added the capacity of the array **elementData**is checked and if it is completely filled that is all element 10 are filled, a new array is created with a new capacity by using **Arrays.copyOf**. If the **elementData**array is not exhausted the new element is added in the array. So adding an element in a array may take more time as a completely new array needs to be created with greater capacity and the data in the old array is transferred into the new array.

The second method adds the element on a specific location in an array.

**add(index i, E element)**  
On adding an element at a particular index in ArrayList, ArrayList checks if an element is already present at that index. If no than the parameter is added at that index, otherwise a new array is created with the index kept vacant and the remaining element shifted to right.

**\*Hashmap vs hashtable?**

**HashMap** is not synchronized. it can contain one null key and multiple null values. it is not ?thread-safe,? so it is useful for non-threaded applications. it inherits the AbstractMap class

**Hashtable** is synchronized. it cannot contain any null key or null value. it is thread-safe, and it can be shared between various threads. it inherits the Dictionary class.

**\*How to synchronize hashmap?**

[HashMap is a non-synchronized collection class](https://beginnersbook.com/2013/12/hashmap-in-java-with-example/). If we need to perform thread-safe operations on it then we must need to synchronize it explicitly. In this tutorial we will see how to synchronize HashMap.

In this example we have a HashMap<Integer, String> it is having integer keys and String type values. In order to synchronize it we are using [Collections.synchronizedMap(hashmap)](https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html" \l "synchronizedMap(java.util.Map)" \t "_blank)  it returns a thread-safe map backed up by the specified HashMap.

**Important point to note in the below example:**  
Iterator should be used in a synchronized block even if we have synchronized the HashMap explicitly (As we did in the below code).

Map map = Collections.synchronizedMap(new HashMap());

...

//This doesn't need to be in synchronized block

Set set = map.keySet();

// Synchronizing on map, not on set

synchronized (map) {

// Iterator must be in synchronized block

Iterator iterator = set.iterator();

while (iterator.hasNext()){

...

}

}

**\*How to synchronize arrayList?**

In order to get a synchronized list from an ArrayList, we use the **synchronizedList(List <T>)** method in Java. The **Collections.synchronizedList(List <T>)** method accepts the ArrayList as an argument and returns a thread safe list.

public static List <T> synchronizedList(List <T> list)

**\*How to iterate map?**

**1. Iterating over Map.entrySet() using For-Each loop :**

**for (Map.Entry<String,String> entry : gfg.entrySet())**

**System.out.println("Key = " + entry.getKey() +**

**", Value = " + entry.getValue());**

**}**

**2. Iterating over keys or values using keySet() and values() methods.**

**// using keySet() for iteration over keys**

**for (String name : gfg.keySet())**

**System.out.println("key: " + name);**

**// using values() for iteration over values**

**for (String url : gfg.values())**

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

**3. Iterating using iterators over [Map.Entry<K, V>](https://docs.oracle.com/javase/7/docs/api/java/util/Map.Entry.html" \t "_blank)**

**// using iterators**

**Iterator<Map.Entry<String, String>> itr = gfg.entrySet().iterator();**

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

**Map.Entry<String, String> entry = itr.next();**

**System.out.println("Key = " + entry.getKey() +**

**", Value = " + entry.getValue());**

**}**

**4. Using forEach(action) method :**

**// forEach(action) method to iterate map**

**gfg.forEach((k,v) -> System.out.println("Key = "**

**+ k + ", Value = " + v));**

**5. Iterating over keys and searching for values (inefficient)**

**// looping over keys**

**for (String name : gfg.keySet()){**

**// search  for value**

**String url = gfg.get(name);**

**System.out.println("Key = " + name + ", Value = " + url);**

**}**

**\*Comparable vs comparator?**

**Comparable:**

1. **Comparable** provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.
2. Comparable **affects the original class**, i.e., the actual class is modified.
3. Comparable provides **compareTo() method** to sort elements.
4. Comparable is present in **java.lang** package.
5. We can sort the list elements of Comparable type by **Collections.sort(List)** method.

**Comparator:**

1. The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2. Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3. Comparator provides **compare() method** to sort elements.
4. A Comparator is present in the **java.util** package.
5. We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

**\*Which method used by comparator internally?**

**Internally** the **Sort** **method** does call Compare **method** of the classes it is sorting. To compare two elements, it asks “Which is greater?” Compare **method** returns -1, 0, or 1 to say if it is less than, equal, or greater to the other.

**\*Hashcode and equals methods?**

Java Object **hashCode**() is a native method and returns the integer hash code value of the object. The general contract of **hashCode**() method is: ... An object hash code value can change in multiple executions of the same application. If two objects are equal according to **equals**() method, then their hash code must be same.

In **HashMap**, **hashCode()** is **used** to calculate the bucket and therefore calculate the index.

**Equals method** is **used** to check that 2 objects are **equal** or not. This **method** is provided by Object class. You can override this in your class to provide your own implementation.

If o1.equals(o2), then o1.hashCode() == o2.hashCode() should always be true.

If o1.hashCode() == o2.hashCode is true, it doesn’t mean that o1.equals(o2) will be true.

Java hashCode() and equals() method are used in Hash table based implementations in java for storing and retrieving data

**\*Arraylist vs vector?**

**ArrayList:**

ArrayList is **not synchronized**.

ArrayList **increments 50%** of current array size if the number of elements exceeds from its capacity.

ArrayList is **not a legacy** class. It is introduced in JDK 1.2.

ArrayList is **fast** because it is non-synchronized.

ArrayList uses the **Iterator** interface to traverse the elements.

**Vector:**

Vector is **synchronized**.

Vector **increments 100%** means doubles the array size if the total number of elements exceeds than its capacity.

Vector is a **legacy** class.

Vector is **slow** because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object.

A Vector can use the **Iterator** interface or **Enumeration** interface to traverse the elements.

**\*How to remove duplicates from arrayList?**

The **[LinkedHashSet](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html" \t "_blank)** is the best approach for removing duplicate elements in an arraylist. LinkedHashSet does two things internally :

Remove duplicate elements and maintain the order of elements added to it.

**// ArrayList with duplicate elements**

**ArrayList<Integer> numbersList = new ArrayList<>(Arrays.asList(1, 1, 2, 3, 3, 3, 4, 5, 6, 6, 6, 7, 8));**

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

**LinkedHashSet<Integer> hashSet = new LinkedHashSet<>(numbersList);**

**ArrayList<Integer> listWithoutDuplicates = new ArrayList<>(hashSet);**

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

To remove the duplicates from the arraylist, we can use the **java 8 stream api** as well. Use steam’s distinct() method which returns a stream consisting of the distinct elements comparing by object’s **equals()** method.

Collect all district elements as List using [Collectors.toList()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html" \l "toList--" \t "_blank).

**// ArrayList with duplicate elements**

**ArrayList<Integer> numbersList = new ArrayList<>(Arrays.asList(1, 1, 2, 3, 3, 3, 4, 5, 6, 6, 6, 7, 8));**

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

**List<Integer> listWithoutDuplicates = numbersList.stream().distinct().collect(Collectors.toList());**

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

**\*What are the cursors used in collection?**

**Enumeration**. **Used** to retrieve elements one by one from a **collection**.

**Iterator.** Iterators in java enable us to retrieve elements one by one.

What is **ListIterator**? An iterator for lists that allows the programmer to traverse the list in either direction.

**\*Difference between arrayBlockingQueue and LinkedBlockingQueue?**

**ArrayBlockingQueue** is a class in Java that implements the **BlockingQueue** interface. ArrayBlockingQueue class and its iterator implement all the optional methods of the Collection and Iterator interfaces.

ArrayBlockingQueue stores the elements in the Queue internally in the **FIFO** **(first-in-first-out)**order.

ArrayBlockingQueue is a bounded BlockingQueue backed by an array.

**// define capacity of ArrayBlockingQueue**

**int capacity = 15;**

**// create object of ArrayBlockingQueue**

**// using ArrayBlockingQueue(int initialCapacity)**

**// constructor**

**ArrayBlockingQueue<Integer> abq**

**= new ArrayBlockingQueue<Integer>(capacity);**

**// add numbers**

**abq.add(1);**

**abq.add(2);**

**abq.add(3);**

**// print queue**

**System.out.println("ArrayBlockingQueue:" + abq);**

[**LinkedBlockingQueue**](https://www.geeksforgeeks.org/linkedblockingqueue-class-in-java/#:~:text=LinkedBlockingQueue%20is%20an%20optionally%2Dbounded,to%20the%20constructor%20of%20LinkedBlockingQueue.):LinkedBlockingQueue is a class in Java that implements the BlockingQueue interface. LinkedBlockingQueue is an **optionally-bounded** BlockingQueue backed by linked nodes.

LinkedBlockingQueue stores the elements in the Queue internally in the **FIFO** **(first-in-first-out)**order.

**// define capacity of LinkedBlockingQueue**

**int capacity = 15;**

**// create object of LinkedBlockingQueue**

**LinkedBlockingQueue<Integer> lbq**

**= new LinkedBlockingQueue<Integer>(capacity);**

**// add numbers**

**lbq.add(1);**

**lbq.add(2);**

**lbq.add(3);**

**// print queue**

**System.out.println("LinkedBlockingQueue:" + lbq);**