Collections in Java

<https://www.javacodegeeks.com/2013/02/40-java-collections-interview-questions-and-answers.html>

A Collection is a group of individual objects represented as a single unit. Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.

The Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are the two main “root” interfaces of Java collection classes.

**Need for Collection Framework :**  
Before Collection Framework (or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were Arrays or Vectors or Hashtables. All of these collections had no common interface.

Accessing elements of these Data Structures was a hassle as each had a different method (and syntax) for accessing its members:

// Java program to show why collection framework was needed

import java.io.\*;

import java.util.\*;

class Test

{

    public static void main (String[] args)

    {

        // Creating instances of array, vector and hashtable

        int arr[] = new int[] {1, 2, 3, 4};

        Vector<Integer> v = new Vector();

        Hashtable<Integer, String> h = new Hashtable();

        v.addElement(1);

        v.addElement(2);

        h.put(1,"geeks");

        h.put(2,"4geeks");

        // Array instance creation requires [], while Vector

        // and hastable require ()

        // Vector element insertion requires addElement(), but

        // hashtable element insertion requires put()

        // Accessing first element of array, vector and hashtable

        System.out.println(arr[0]);

        System.out.println(v.elementAt(0));

        System.out.println(h.get(1));

        // Array elements are accessed using [], vector elements

        // using elementAt() and hashtable elements using get()

    }

}

Output:

1

1

geeks

As we can see, none of these collections (Array, Vector or Hashtable) implement a standard member access interface. It was very difficult for programmers to write algorithms that can work for all kinds of Collections. Another drawback being that most of the ‘Vector’ methods are final, meaning we cannot extend the ’Vector’ class to implement a similar kind of Collection.  
***Java developers decided to come up with a common interface to deal with the above mentioned problems and introduced the Collection Framework in JDK 1.2***.

Both legacy Vectors and Hashtables were modified to conform to the Collection Framework.

**Advantages of Collection Framework:**

1. Consistent API : The API has a basic set of interfaces like Collection, Set, List, or Map. All classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have *some* common set of methods.
2. Reduces programming effort: A programmer doesn’t have to worry about the design of Collection, and he can focus on its best use in his program.
3. Increases program speed and quality: Increases performance by providing high-performance implementations of useful data structures and algorithms.

**Hierarchy of Collection Framework**

Collection Map

/ / \ \ |

/ / \ \ |

Set List Queue Dequeue SortedMap

/

/

SortedSet

**Core Interfaces in Collections**

Note that this diagram only shows core interfaces.

**Collection :** Root interface with basic methods like add(), remove(),

contains(), isEmpty(), addAll(), ... etc.

[**Set**](https://www.geeksforgeeks.org/set-in-java/) **:** Doesn't allow duplicates. Example implementations of Set

interface are HashSet (Hashing based) and TreeSet (balanced

BST based). Note that TreeSet implements **SortedSet**.

[**List**](https://www.geeksforgeeks.org/list-interface-java-examples/) **:** Can contain duplicates and elements are ordered. Example

implementations are LinkedList (linked list based) and

[ArrayList](https://www.geeksforgeeks.org/array-vs-arraylist-in-java/) (dynamic array based)

[**Queue**](https://www.geeksforgeeks.org/queue-interface-java/) **:** Typically order elements in FIFO order except exceptions

like PriorityQueue.

[**Deque**](https://www.geeksforgeeks.org/deque-interface-java-example/) **:** Elements can be inserted and removed at both ends. Allows

both LIFO and FIFO.

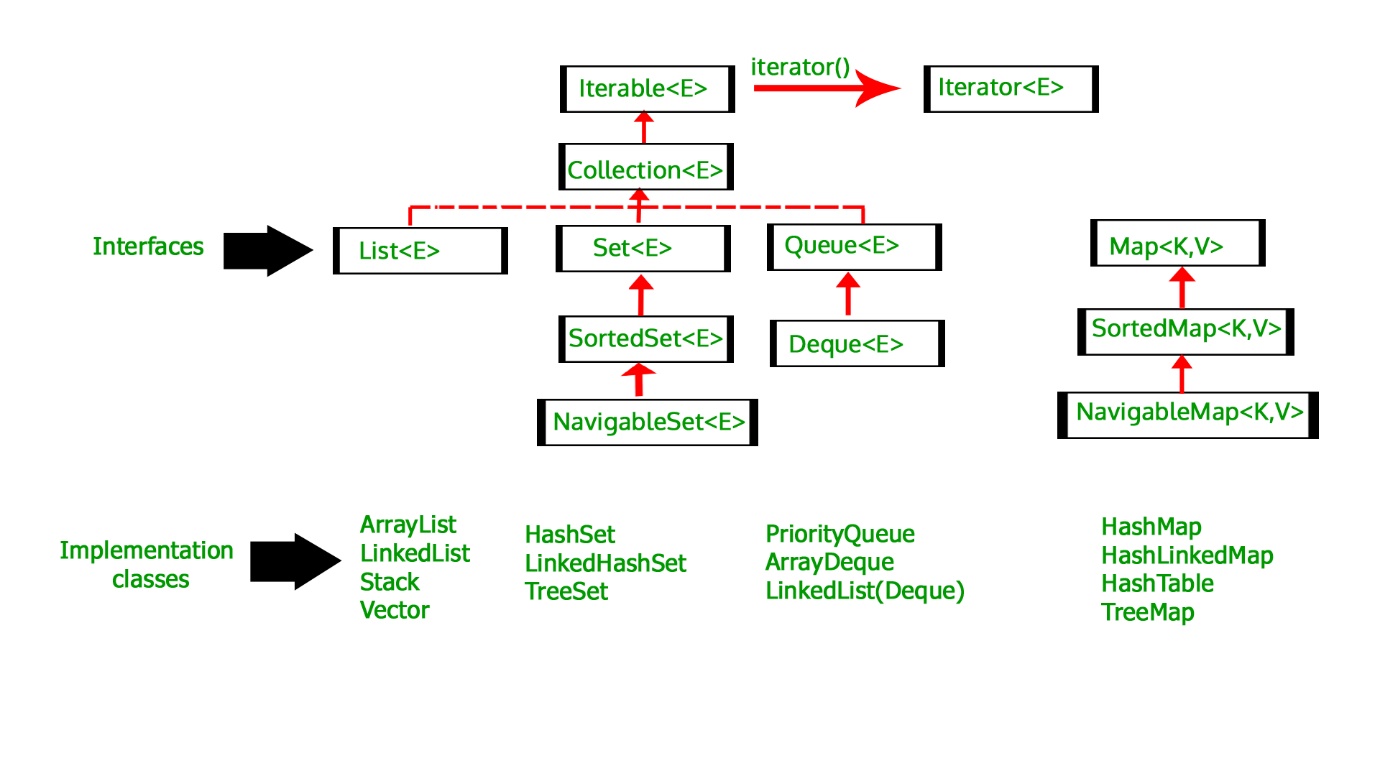
[**Map**](https://www.geeksforgeeks.org/map-interface-java-examples/) **:** Contains Key value pairs. Doesn't allow duplicates. Example

implementation are [HashMap](http://www.geeksforgeeks.org/java-util-hashmap-in-java/) and TreeMap.

[TreeMap](https://www.geeksforgeeks.org/treemap-in-java/) implements **SortedMap**.

The difference between Set and Map interface is that in Set we

have only keys, whereas in Map, we have key, value pairs.



# Comparable vs Comparator in Java

Java provides two interfaces to sort objects using data members of the class:

1. Comparable
2. Comparator

**Using Comparable Interface**

A comparable object is capable of comparing itself with another object. The class itself must implements the **java.lang.Comparable** interface to compare its instances.

Consider a Movie class that has members like, rating, name, year. Suppose we wish to sort a list of Movies based on year of release. We can implement the Comparable interface with the Movie class, and we override the method compareTo() of Comparable interface.

**Using Comparator**

Unlike Comparable, Comparator is external to the element type we are comparing. It’s a separate class. We create multiple separate classes (that implement Comparator) to compare by different members.

Collections class has a second sort() method and it takes Comparator. The sort() method invokes the compare() to sort objects.

To compare movies by Rating, we need to do 3 things :

1. Create a class that implements Comparator (and thus the compare() method that does the work previously done by compareTo()).
2. Make an instance of the Comparator class.
3. Call the overloaded sort() method, giving it both the list and the instance of the class that implements Comparator.

* Comparable is meant for objects with natural ordering which means the object itself must know how it is to be ordered. For example Roll Numbers of students. Whereas, Comparator interface sorting is done through a separate class.
* Logically, Comparable interface compares “this” reference with the object specified and Comparator in Java compares two different class objects provided.
* If any class implements Comparable interface in Java then collection of that object either List or Array can be sorted automatically by using Collections.sort() or Arrays.sort() method and objects will be sorted based on there natural order defined by CompareTo method.

***To summarize, if sorting of objects needs to be based on natural order then use Comparable whereas if you sorting needs to be done on attributes of different objects, then use Comparator in Java.***

**There are five ways to traverse collections:**

**1. By using Iterator**

**2. By using ListIterator**

**3. By using Enumeration**

**4. By using for-each**

**5. By using for loop**

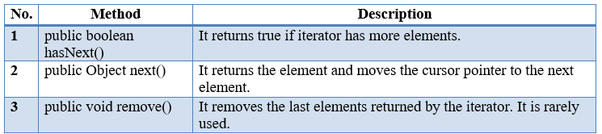
**1 . Iterator interface :**

Iterator interface used to traverse the elements in forward direction only.

*Iterator* which is used to traverse most of the classes in the collection framework including legacy classes. Iterator can be used in **List, Set , Queue, and Vector etc.**

You get an Iterator for a collection by calling its **iterator(**) method.

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



1. List<String> obj =**new** ArrayList<String>();
2. obj.add("Mango");
3. obj.add("Apple");
5. Iterator<String> ite=obj.iterator(); //obj is collection object
6. **while**(ite.hasNext())//It returns true if iterator has more elements.
7. {
8. System.**out**.println(ite.**next**()); //print the value
9. }

**2. ListIterator interface :**

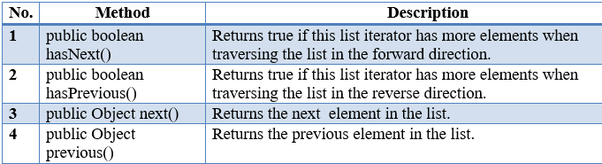
ListIterator interface extends Iterator interface.

ListIterator interface used to traverse the elements in both directions (forward and backward), but reverse apply only for LinkedList elements only.

ListIterator can be used to traverse for **List-type Objects**, but not for Set-type of Objects.

You get a ListIterator for a collection by calling its **listIterator()** method.

**There are many methods in the ListIterator interface. But commonly used listed below:**



1. List<String> obj=**new** LinkedList<String>();
2. obj.add("Mango");
3. obj.add("Apple");
4. obj.add("Orange");
6. ListIterator lte=obj.listIterator(); //obj is LinkedList object
7. **while**(lte.hasPrevious()) // has previous supports only LinkedList
8. {
9. System.**out**.println(" "+lte.previous()); //print reverse order
10. }

**3. Enumeration interface :**

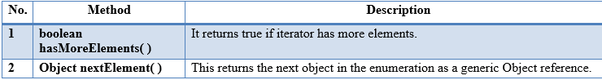
The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. It supports most of the collections objects.

*Enumeration* is a **legacy interface** used to **traverse only the legacy classes like *Vector*, *HashTable* and *Stack*.**

There is only one legacy interface called **Enumeration.**

You get an Enumeration for a collection by calling its **elements()** method.

**There are two methods in the Enumeration interface. They are:**



1. Vector obj=**new** Vector();
2. obj.add("Mango");
3. obj.add("Apple");
5. Enumeration<String> enu=obj.elements(); //obj is Vector object
6. **while**(enu.hasMoreElements())
7. {
8. System.**out**.println(" "+ enu.nextElement());
9. }

**4. By using for-each**

It is mainly used to traverse array or collection elements. The advantage of for-each loop is that it eliminates the possibility of bugs and makes the code more readable.

Please refer below example so that you can understand better.

**Example:**

**traverse collection using for-each:**

1. List<String> obj =**new** ArrayList<String>();
2. obj.add("Mango");
3. obj.add("Apple");
5. **for**(String kk:obj) ////obj is collection object
6. {
7. System.**out**.println("Elements :"+kk);
8. }

**traverse array using for-each:**

1. **int** arr[]={12,13,14,44};
3. **for**(**int** i:arr){
4. System.**out**.println(i);
5. }

**5. By using for loop**

**Example:**

1. LinkedList obj=**new** LinkedList();
2. obj.add("Mango");
3. obj.add("Apple");
4. obj.add("Banana");
6. **for**(**int** i=0;i<obj.size();i++) {
7. System.**out**.println(" "+obj.**get**(i));
8. }
9. **HashMap:** HashMap offers **0(1)** lookup and insertion. If you iterate through the keys, though, the ordering of the keys is essentially arbitrary. It is implemented by an array of linked lists.  
   **Syntax:**

**public class HashMap extends AbstractMap**

**implements Map,Cloneable, Serializable**

* + A HashMap contains values based on the key.
  + It contains only unique elements.
  + It may have one null key and multiple null values.
  + It maintains **no order**.

1. **LinkedHashMap:**LinkedHashMap offers **0(1)** lookup and insertion. Keys are ordered by their insertion order. It is implemented by doubly-linked buckets.  
   **Syntax:**

**public class LinkedHashMap extends HashMap**

**0implements Map**

* + A LinkedHashMap contains values based on the key.
  + It contains only unique elements.
  + It may have one null key and multiple null values.
  + It is same as HashMap instead **maintains insertion order**.

1. **TreeMap:** TreeMap offers **O(log N)** lookup and insertion. Keys are ordered, so if you need to iterate through the keys in sorted order, you can. This means that keys must implement the Comparable interface. TreeMap is implemented by a Red-Black Tree.  
   **Syntax:**

**public class TreeMap extends AbstractMap implements**

**NavigableMap, Cloneable, Serializable**

* + A TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
  + It contains only unique elements.
  + It cannot have null key but can have multiple null values.
  + It is same as HashMap instead **maintains ascending order(Sorted using the natural order of its key**).

1. **Hashtable:**“Hashtable” is the generic name for hash-based maps.  
   **Syntax:**

**public class Hashtable extends Dictionary implements**

**Map, Cloneable, Serializable**

* + A Hashtable is an array of list. Each list is known as a bucket. The position of bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
  + It contains only unique elements.
  + It may have not have any null key or value.
  + It is synchronized.
  + It is a legacy class

# ConcurrentHashMap in java

**ConcurrentHashMap** ConcurrentHashMap class is introduced in JDK 1.5, which implements ConcurrentMap as well as Serializable interface also. ConcureentHashMap 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 upto the mark.

**Key points of ConcurrentHashMap:**

It may have not have any null key or value.

* 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.

**Constructors of ConcurrentHashMap:**

1. **ConcurrentHashMap m=new ConcurrentHashMap();**:Creates a new, empty map with a default initial capacity (16), load factor (0.75) and concurrencyLevel (16).
2. **ConcurrentHashMap m=new ConcurrentHashMap(int initialCapacity);**:Creates a new, empty map with the specified initial capacity, and with default load factor (0.75) and concurrencyLevel (16).
3. **ConcurrentHashMap m=new ConcurrentHashMap(int initialCapacity, float loadFactor);**:  
   Creates a new, empty map with the specified initial capacity and load factor and with the default concurrencyLevel (16).
4. **ConcurrentHashMap m=new ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel);**:Creates a new, empty map with the specified initial capacity, load factor and concurrency level.
5. **ConcurrentHashMap m=new ConcurrentHashMap(Map m);**:Creates a new map with the same mappings as the given map.

// Java program to demonstrate working of ConcurrentHashMap

import java.util.concurrent.\*;

class ConcurrentHashMapDemo {

    public static void main(String[] args)

    {

        ConcurrentHashMap m = new ConcurrentHashMap();

        m.put(100, "Hello");

        m.put(101, "Geeks");

        m.put(102, "Geeks");

        // Here we cant add Hello because 101 key

        // is already present in ConcurrentHashMap object

        m.putIfAbsent(101, "Hello");

        // We can remove entry because 101 key

        // is associated with For value

        m.remove(101, "Geeks");

        // Now we can add Hello

        m.putIfAbsent(103, "Hello");

        // We cant replace Hello with For

        m.replace(101, "Hello", "For");

        System.out.println(m);

    }

}

Output:

{100=Hello, 102=Geeks, 103=Hello}

Notes :

* Hashtable is belongs to the Collection framework; ConcurrentHashMap belongs to the Executor framework.
* Hashtable uses **single lock** for whole data. ConcurrentHashMap uses **multiple locks** on segment level (16 by default) instead of object level i.e. whole Map.
* ConcurrentHashMap locking is applied only for updates. In case of retrievals, it allows full concurrency, retrievals reflect the results of the most recently completed update operations. So reads can happen very fast while writes are done with a lock.
* ConcurrentHashMap doesn't throw a ConcurrentModificationException if one thread tries to modify it while another is iterating over it and does not allow null values.
* ConcurrentHashMap returns Iterator, which fails-safe (i.e. iterator will make a copy of the internal data structure) on concurrent modification.
* ConcurrentHashMap uses a database shards logic (Segment<K, V>[] segments) is known as **Concurrency-Level**, i.e. divides the data into shards(segments) than puts locks on each shard (segment) instead of putting a single lock for whole data (Map). The default value is 16.

**The following analogy helps you get understand the concept only(not logic)**

* Assume Hashtable and ConcurrentHashMap are two types of Homes.
* Hashtable locks home's main door.
* ConcurrentHashMap locks specific room door instead of main door.

# Naval need to write worked on each collection in detailed time 06JUNE 5:35PM