Collections Interview Questions and Answers

Q. What is Java Collections Framework? List out some benefits of Collections framework?

Java Collections Framework

The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects. Java Collections can achieve all the operations that we perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

1. Collection Interface

Collection interface is at the root of the hierarchy. Collection interface provides all general purpose methods which all collections classes must support (or throw UnsupportedOperationException). It extends **Iterable** interface which adds support for iterating over collection elements using the "for-each loop" statement.

2. List

Lists represents an **ordered collection** of elements. Using lists, we can access elements by their integer index (position in the list), and search for elements in the list. index start with 0, just like an array.

Some useful classes which implement List interface are – **ArrayList**, **CopyOnWriteArrayList**, **LinkedList**, **Stack** and **Vector**.

3. Set

Sets represents a collection of **sorted** elements. Sets do not allow the duplicate elements. Set interface does not provides no guarantee to return the elements in any predictable order; though some Set implementations store elements in their natural ordering and guarantee this order.

Some useful classes which implement Set interface are – **ConcurrentSkipListSet**, **CopyOnWriteArraySet**, **EnumSet**, **HashSet**, **LinkedHashSet** and **TreeSet**.

4. Map

The Map interface enable us to store data in key-value pairs (keys should be immutable). A map cannot contain duplicate keys; each key can map to at most one value.

The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. Some map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not.

Some useful classes which implement Map interface are – **ConcurrentHashMap**, **ConcurrentSkipListMap**, **EnumMap**, **HashMap**, **Hashtable**, **IdentityHashMap**, **LinkedHashMap**, **Properties**, **TreeMap** and **WeakHashMap**.

5. Stack

The Java Stack interface represents a classical stack data structure, where elements can be pushed to last-in-first-out (LIFO) stack of objects. In Stack we push an element to the top of the stack, and popped off from the top of the stack again later.

6. Queue

A queue data structure is intended to hold the elements (put by producer threads) prior to processing by consumer thread(s). Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations.

Some useful classes which implement Map interface are – **ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue and SynchronousQueue**.

7. Deque

A double ended queue (pronounced "deck") that supports element insertion and removal at both ends. When a deque is used as a queue, FIFO (First-In-First-Out) behavior results. When a deque is used as a stack, LIFO (Last-In-First-Out) behavior results.

Some common known classes implementing this interface are **ArrayDeque, ConcurrentLinkedDeque, LinkedBlockingDeque** and **LinkedList**.

	List	Set	Queue	Map
Order	Yes	No	Yes	No
Duplicates	Yes	No	Yes	No (Allow duplicate values not keys)
Null Values	Yes	Single Null	Yes (LinkedList Queue). No (Priority Queue).	Single null key and many null values

The Java Collections Framework provides the following benefits:

- Reduces programming effort
- Increases program speed and quality
- Allows interoperability among unrelated APIs
- Reduces effort to learn and to use new APIs
- Reduces effort to design new APIs
- Fosters software reuse

Methods of Collection Interface

No.	Method	Description
1	public boolean add(Object element)	is used to insert an element in this collection.
2	public boolean addAll(Collection c)	is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	is used to delete an element from this collection.
4	public boolean removeAll(Collection c)	is used to delete all the elements of specified collection from the invoking collection.
5	public boolean retainAll(Collection c)	is used to delete all the elements of invoking collection except the specified collection.
6	public int size()	return the total number of elements in the collection.
7	public void clear()	removes the total no of element from the collection.
8	public boolean contains(Object element)	is used to search an element.
9	public boolean containsAll(Collection c)	is used to search the specified collection in this collection.
10	public Iterator iterator()	returns an iterator.
11	public Object[] toArray()	converts collection into array.
12	public boolean isEmpty()	checks if collection is empty.
13	public boolean equals(Object element)	matches two collection.
14	public int hashCode()	returns the hashcode number for collection.

Collections Framework Implementation Classes Summary

Collection Class

↥ back to top

Q. What will be the problem if you do not override hashcode() method?

Some collections, like HashSet, HashMap or HashTable use the hashcode value of an object to find out how the object would be stored in the collection, and subsequently hashcode is used to help locate the object in the collection. Hashing retrieval involves:

- First, find out the right bucket using hashCode().
- Secondly, search the bucket for the right element using equals()

If hashcode() in not overridden then the default implementation in Object class will be used by collections. This implementation gives different values for different objects, even if they are equal according to the equals() method.

Example:

```
public class Student {
    private int id;
    private String name;
    public Student(int id, String name) {
        this.name = name;
        this.id = id;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}

public class HashcodeEquals {
    public static void main(String[] args) {
        Student alex1 = new Student(1, "Alex");
        Student alex2 = new Student(1, "Alex");
        System.out.println("alex1 hashcode = " + alex1.hashCode());
        System.out.println("alex2 hashcode = " + alex2.hashCode());
        System.out.println("Checking equality between alex1 and alex2 = " + alex1.equals(alex2));
    }
}
```

Output

```
alex1 hashcode = 1852704110
alex2 hashcode = 2032578917
Checking equality between alex1 and alex2 = false
```

↥ back to top

Q. What is the benefit of Generics in Collections Framework?

Generics allow us to provide the type of Object that a collection can contain, so if we try to add any element of other type it throws compile time error. This avoids ClassCastException at Runtime because we will get the error at compilation. Also Generics make code clean since we don't need to use casting and instanceof operator.

Q. How do WeakHashMap works?

WeakHashMap is a Hash table-based implementation of the Map interface with weak keys. An entry in a WeakHashMap will automatically be removed when its key is no longer in ordinary use. Both null values and the null key are supported. This class has performance characteristics similar to those of the HashMap class and has the same efficiency parameters of initial capacity and load factor.

```
// Java program to illustrate
// WeakHashmap
import java.util.Map;
import java.util.Map.Entry;
import java.util.WeakHashMap;
public class WeakHashMapExample {

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

        final Map map = new WeakHashMap<>();
        Key key1 = new Key("ACTIVE");
        final Key key2 = new Key("INACTIVE");
        map.put(key1, new Project(100, "Customer Management System", "Customer Management System"));
        map.put(key2, new Project(200, "Employee Management System", "Employee Management System"));
        key1 = null;
        System.gc();
        for (final Entry entry : map.entrySet()) {
            System.out.println(entry.getKey().getKey() + "   " + entry.getValue());
        }
    }
}
class Key {
    private String key;
    public Key(final String key) {
        super();
        this.key = key;
    }
    public String getKey() {
        return key;
    }
    public void setKey(final String key) {
        this.key = key;
    }
}
```

Output

```
INACTIVE   [project id : 200, project name : Employee Management System,
           project desc : Employee Management System ]
```

↥ back to top

Q. What is difference between Array and ArrayList?

1. Size: Array in Java is fixed in size. We can not change the size of array after creating it. ArrayList is dynamic in size. When we add elements to an ArrayList, its capacity increases automatically.

2. Performance: In Java Array and ArrayList give different performance for different operations.

add() or get(): Adding an element to or retrieving an element from an array or ArrayList object has similar performance. These are constant time operations.

resize(): Automatic resize of ArrayList slows down the performance. ArrayList is internally backed by an Array. In resize() a temporary array is used to copy elements from old array to new array.

3. Primitives: Array can contain both primitive data types as well as objects. But ArrayList can not contain primitive data types. It contains only objects.

4. Iterator: In an ArrayList we use an Iterator object to traverse the elements. We use for loop for iterating elements in an array.

5. Type Safety: Java helps in ensuring Type Safety of elements in an ArrayList by using Generics. An Array can contain objects of same type of classe. If we try to store a different data type object in an Array then it throws ArrayStoreException.

6. Length: Size of ArrayList can be obtained by using size() method. Every array object has length variable that is same as the length/size of the array.

7. Adding elements: In an ArrayList we can use add() method to add objects. In an Array assignment operator is used for adding elements.

8. Multi-dimension: An Array can be multi-dimensional. An ArrayList is always of single dimension

```
// A Java program to demonstrate differences between array
// and ArrayList
import java.util.ArrayList;
import java.util.Arrays;

class Test
{
    public static void main(String args[]) {
        /* ........... Normal Array.............. */
        int[] arr = new int[2];
        arr[0] = 10;
        arr[1] = 20;
        System.out.println(arr[0]);

        /*............ArrayList..............*/
        // Create an arrayList with initial capacity 2
        ArrayList arrL = new ArrayList(2);

        // Add elements to ArrayList
        arrL.add(30);
        arrL.add(40);

        // Access elements of ArrayList
        System.out.println(arrL.get(0));
    }
}
```

Output

```
10
30
```

↥ back to top

Q. What is difference between ArrayList and LinkedList?

ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.

Sl.No	ArrayList	LinkedList
01.	ArrayList internally uses a dynamic array to store the elements.	LinkedList internally uses a doubly linked list to store the elements.
02.	Manipulation with ArrayList is slow because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
03.	An ArrayList class can act as a list only because it implements List only.	LinkedList class can act as a list and queue both because it implements List and Deque interfaces.
04.	ArrayList is better for storing and accessing data.	LinkedList is better for manipulating data.

```
// Java program to demonstrate difference between ArrayList and
// LinkedList.
import java.util.ArrayList;
import java.util.LinkedList;

public class ArrayListLinkedListExample
{
 public static void main(String[] args) {

   ArrayList arrlistobj = new ArrayList();
   arrlistobj.add("One");
   arrlistobj.add("Two");
   arrlistobj.add("Three");
   arrlistobj.remove(1); // Remove value at index 2
   System.out.println("ArrayList object output: " + arrlistobj);

   // Checking if an element is present.
```

```
   if (arrlistobj.contains("Two"))
    System.out.println("Found");
   else
    System.out.println("Not found");


   LinkedList llobj = new LinkedList();
   llobj.add("Four");
   llobj.add("Five");
   llobj.add("Six");
   llobj.remove("Five");
   System.out.println("LinkedList object output: " + llobj);

   // Checking if an element is present.
   if (llobj.contains("Five"))
    System.out.println("Found");
   else
    System.out.println("Not found");
  }
}
```

↥ back to top

Q. What is difference between Comparable and Comparator interface?

Comparable: A comparable object is capable of comparing itself with another object. The class itself must implements the `java.lang.Comparable` interface in order to be able to compare its instances.

Comparator: A comparator object is capable of comparing two different objects. The class is not comparing its instances, but some other class's instances. This comparator class must implement the `java.util.Comparator` interface.

Comparable and Comparator both are interfaces and can be used to sort collection elements.

Sl.No	Comparable	Comparator
01.	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.	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.
02.	Comparable affects the original class, i.e., the actual class is modified.	Comparator doesn't affect the original class, i.e., the actual class is not modified.
03.	Comparable provides compareTo() method to sort elements.	Comparator provides compare() method to sort elements.
04.	Comparable is present in java.lang package.	A Comparator is present in the java.util package.
05.	We can sort the list elements of Comparable type by Collections.sort(List) method.	We can sort the list elements of Comparator type by Collections.sort(List, Comparator) method.

```
//Java Program to demonstrate the use of Java Comparable.
//Creating a class which implements Comparable Interface
import java.util.*;
import java.io.*;

class Student implements Comparable {
  int rollno;
  String name;
  int age;
  Student(int rollno,String name,int age) {
    this.rollno = rollno;
    this.name = name;
    this.age = age;
  }
  public int compareTo(Student st){
    if(age == st.age)
        return 0;
    else if(age > st.age)
        return 1;
    else
        return -1;
  }
}
//Creating a test class to sort the elements
public class TestSort3 {
```

```
  public static void main(String args[]) {
    ArrayList al = new ArrayList();
    al.add(new Student(101,"Vijay",23));
    al.add(new Student(106,"Ajay",27));
    al.add(new Student(105,"Jai",21));

    Collections.sort(al);
    for(Student st:al) {
        System.out.println(st.rollno+" "+st.name+" "+st.age);
    }
  }
}
```

↥ back to top

Q. How to remove duplicates from ArrayList?

The LinkedHashSet is the best approach for removing duplicate elements in an arraylist. LinkedHashSet does two things internally :

- Remove duplicate elements
- Maintain the order of elements added to it

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedHashSet;

public class ArrayListExample
{
    public static void main(String[] args) {

        // ArrayList with duplicate elements
        ArrayList numbersList = new ArrayList<>(Arrays.asList(1, 1, 2, 3, 3, 3, 4, 5, 6, 6, 6, 7, 8));

        System.out.println("ArrayList with duplicate elements: ", numbersList);

        LinkedHashSet hashSet = new LinkedHashSet<>(numbersList);
        ArrayList listWithoutDuplicates = new ArrayList<>(hashSet);

        System.out.println("ArrayList without duplicate elements: ", listWithoutDuplicates);
    }
}
```

Output

```
ArrayList with duplicate elements: [1, 1, 2, 3, 3, 3, 4, 5, 6, 6, 6, 7, 8]
ArrayList without duplicate elements: [1, 2, 3, 4, 5, 6, 7, 8]
```

Q. What is Java Priority Queue?

A priority queue in Java is a special type of queue wherein all the elements are ordered as per their natural ordering or based on a custom Comparator supplied at the time of creation.

The front of the priority queue contains the least element according to the specified ordering, and the rear of the priority queue contains the greatest element. So when we remove an element from the priority queue, the least element according to the specified ordering is removed first. The Priority Queue class is part of Java's collections framework and implements the Queue interface.

features

- PriorityQueue is an unbounded queue and grows dynamically.
- It does not allow NULL objects.
- Objects added to PriorityQueue MUST be comparable.
- The objects of the priority queue are ordered **by default in natural order**.
- A Comparator can be used for custom ordering of objects in the queue.
- The **head** of the priority queue is the **least** element based on the natural ordering or comparator based ordering. When we poll the queue, it returns the head object from the queue.
- If multiple objects are present of same priority the it can poll any one of them randomly.

- PriorityQueue is **not thread safe**. Use `PriorityBlockingQueue` in concurrent environment.
- It provides **O(log(n))** time for add and poll methods.

```
import java.util.*;

public class CreatePriorityQueueStringExample {

    public static void main(String args[]) {

        PriorityQueue queue = new PriorityQueue();
        queue.add("Amit");
        queue.add("Vijay");
        queue.add("Karan");
        queue.add("Jai");
        queue.add("Rahul");
        System.out.println("head: "+queue.element());
        System.out.println("head: "+queue.peek());
        System.out.println("Iterating the queue elements: ");

        Iterator itr = queue.iterator();
        while(itr.hasNext()) {
            System.out.println(itr.next());
        }
        queue.remove();
        queue.poll();
        System.out.println("after removing two elements: ");

        Iterator itr2 = queue.iterator();
        while(itr2.hasNext()) {
            System.out.println(itr2.next());
        }
    }
}
```

Output

```
head: Amit
head: Amit
iterating the queue elements:
Amit
Jai
Karan
Vijay
Rahul
after removing two elements:
Karan
Rahul
Vijay
```

↥ back to top

Q. What is LinkedHashMap in Java?

LinkedHashMap is just like HashMap with an additional feature of maintaining an order of elements inserted into it. Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

Features

- Java LinkedHashMap contains values based on the key.
- Java LinkedHashMap contains unique elements.
- Java LinkedHashMap may have one null key and multiple null values.
- Java LinkedHashMap is non synchronized.
- Java LinkedHashMap maintains insertion order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

```
import java.util.LinkedHashMap;
import java.util.Set;
import java.util.Iterator;
import java.util.Map;
public class LinkedHashMapDemo {
    public static void main(String args[]) {
```

```
        // HashMap Declaration
        LinkedHashMap lhmap =
                new LinkedHashMap();
        //Adding elements to LinkedHashMap
        lhmap.put(22, "Abey");
        lhmap.put(33, "Dawn");
        lhmap.put(1, "Sherry");
        lhmap.put(2, "Karon");
        lhmap.put(100, "Jim");

        // Generating a Set of entries
        Set set = lhmap.entrySet();

        // Displaying elements of LinkedHashMap
        Iterator iterator = set.iterator();
        while(iterator.hasNext()) {
           Map.Entry me = (Map.Entry)iterator.next();
           System.out.print("Key is: "+ me.getKey() +
                   "& Value is: "+me.getValue()+"\n");
        }
    }
}
```

Output

```
Key is: 22  & Value is: Abey
Key is: 33  & Value is: Dawn
Key is: 1   & Value is: Sherry
Key is: 2   & Value is: Karon
Key is: 100 & Value is: Jim
```

Q. What are different Collection views provided by Map interface?

Hierarchy of Map Interface

Map Interface

In the inheritance tree of the Map interface, there are several implementations but only 3 major, common, and general purpose implementations - they are HashMap and LinkedHashMap and TreeMap.

1. HashMap

This implementation uses a hash table as the underlying data structure. It implements all of the Map operations and allows null values and one null key. This class is roughly equivalent to Hashtable - a legacy data structure before Java Collections Framework, but it is not synchronized and permits nulls. HashMap does not guarantee the order of its key-value elements. Therefore, consider to use a HashMap when order does not matter and nulls are acceptable.

```
Map mapHttpErrors = new HashMap<>();

mapHttpErrors.put(200, "OK");
mapHttpErrors.put(303, "See Other");
mapHttpErrors.put(404, "Not Found");
mapHttpErrors.put(500, "Internal Server Error");

System.out.println(mapHttpErrors);
```

Output

```
{404=Not Found, 500=Internal Server Error, 200=OK, 303=See Other}
```

2. LinkedHashMap

This implementation uses a hash table and a linked list as the underlying data structures, thus the order of a LinkedHashMap is predictable, with insertion-order as the default order. This implementation also allows nulls like HashMap. So consider using a LinkedHashMap when you want a Map with its key-value pairs are sorted by their insertion order.

```
Map mapContacts = new LinkedHashMap<>();

mapContacts.put("0169238175", "Tom");
mapContacts.put("0904891321", "Peter");
```

```
mapContacts.put("0945678912", "Mary");
mapContacts.put("0981127421", "John");

System.out.println(mapContacts);
```

Output

```
{0169238175=Tom, 0904891321=Peter, 0945678912=Mary, 0981127421=John}
```

3. TreeMap

This implementation uses a red-black tree as the underlying data structure. A TreeMap is sorted according to the natural ordering of its keys, or by a Comparator provided at creation time. This implementation does not allow nulls. So consider using a TreeMap when you want a Map sorts its key-value pairs by the natural order of the keys (e.g. alphabetic order or numeric order), or by a custom order you specify.

```
Map mapLang = new TreeMap<>();

mapLang.put(".c", "C");
mapLang.put(".java", "Java");
mapLang.put(".pl", "Perl");
mapLang.put(".cs", "C#");
mapLang.put(".php", "PHP");
mapLang.put(".cpp", "C++");
mapLang.put(".xml", "XML");

System.out.println(mapLang);
```

Output

```
{.c=C, .cpp=C++, .cs=C#, .java=Java, .php=PHP, .pl=Perl, .xml=XML}
```

Useful Methods of Map Interface

Method	Description
Object put(Object key, Object value)	It is used to insert an entry in this map.
void putAll(Map map)	It is used to insert the specified map in this map.
Object remove(Object key)	It is used to delete an entry for the specified key.
Object get(Object key)	It is used to return the value for the specified key.
boolean containsKey(Object key)	It is used to search the specified key from this map.
Set keySet()	It is used to return the Set view containing all the keys.
Set entrySet()	It is used to return the Set view containing all the keys and values.

Methods of Map.Entry Interface

Method	Description
Object getKey()	It is used to obtain key.
Object getValue()	It is used to obtain value.

↥ back to top

Q. What is difference between HashMap and Hashtable?

HashMap and Hashtable both are used to store data in key and value form. Both are using hashing technique to store unique keys.

Sl.No	HashMap	Hashtable
01.	HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code.	Hashtable is **synchronized**. It is thread-safe and can be shared with many threads.
02.	HashMap allows one null key and multiple null values.	Hashtable doesn't allow any null key or value.

Sl.No	HashMap	Hashtable
03.	HashMap is a new class introduced in JDK 1.2.	Hashtable is a legacy class.
04.	HashMap is fast.	Hashtable is slow.
05.	We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap);	Hashtable is internally synchronized and can't be unsynchronized.
06.	HashMap is traversed by Iterator.	Hashtable is traversed by Enumerator and Iterator.
07.	Iterator in HashMap is fail-fast.	Enumerator in Hashtable is not fail-fast.
08.	HashMap inherits AbstractMap class.	Hashtable inherits Dictionary class.

```
/**
   A sample Java program to demonstrate HashMap and HashTable
**/
import java.util.*;
import java.lang.*;
import java.io.*;

class HashMapHashtableExample
{
    public static void main(String args[]) {

        // hashtable
        Hashtable ht = new Hashtable();
        ht.put(101," ajay");
        ht.put(101,"Vijay");
        ht.put(102,"Ravi");
        ht.put(103,"Rahul");

        System.out.println("Hash table: ");
        for (Map.Entry m:ht.entrySet()) {
            System.out.println(m.getKey()+" "+m.getValue());
        }

        // hashmap
        HashMap hm = new HashMap();
        hm.put(100,"Amit");
        hm.put(104,"Amit");  // hash map allows duplicate values
        hm.put(101,"Vijay");
        hm.put(102,"Rahul");
        System.out.println("Hash map: ");
        for (Map.Entry m:hm.entrySet()) {
            System.out.println(m.getKey()+" "+m.getValue());
        }
    }
}
```

Output

```
Hash table:
103 Rahul
102 Ravi
101 Vijay

Hash map:
100 Amit
101 Vijay
102 Rahul
104 Amit
```


↥ back to top

Q. What is EnumSet?

Java EnumSet class is the specialized Set implementation for use with enum types. It inherits AbstractSet class and implements the Set interface.

Features

- It can contain only enum values and all the values have to belong to the same enum
- It doesn't allow to add null values, throwing a NullPointerException in an attempt to do so

- It's not thread-safe, so we need to synchronize it externally if required
- The elements are stored following the order in which they are declared in the enum
- It uses a fail-safe iterator that works on a copy, so it won't throw a ConcurrentModificationException if the collection is modified when iterating over it

```
import java.util.EnumSet;
import java.util.Set;
/**
* Simple Java Program to demonstrate how to use EnumSet.
* It has some interesting use cases and it's specialized collection for
* Enumeration types. Using Enum with EnumSet will give you far better
* performance than using Enum with HashSet, or LinkedHashSet.
*
**/
public class EnumSetDemo {
    private enum Color {
        RED(255, 0, 0), GREEN(0, 255, 0), BLUE(0, 0, 255);
        private int r;
        private int g;
        private int b;
        private Color(int r, int g, int b) {
            this.r = r; this.g = g; this.b = b;
        }
        public int getR() {
            return r;
        }
        public int getG() {
            return g;
        }
        public int getB() {
            return b;
        }
    }
    public static void main(String args[]) {
        // this will draw line in yellow color
        EnumSet yellow = EnumSet.of(Color.RED, Color.GREEN);
        drawLine(yellow);
        // RED + GREEN + BLUE = WHITE
        EnumSet white = EnumSet.of(Color.RED, Color.GREEN, Color.BLUE);
        drawLine(white);
        // RED + BLUE = PINK
        EnumSet pink = EnumSet.of(Color.RED, Color.BLUE);
        drawLine(pink);
    }
    public static void drawLine(Set colors) {
        System.out.println("Requested Colors to draw lines : " + colors);
        for (Color c : colors) {
            System.out.println("drawing line in color : " + c);
        }
    }
}
```

Output

```
Output: Requested Colors to draw lines : [RED, GREEN]
drawing line in color : RED
drawing line in color : GREEN

Requested Colors to draw lines : [RED, GREEN, BLUE]
drawing line in color : RED
drawing line in color : GREEN
drawing line in color : BLUE

Requested Colors to draw lines : [RED, BLUE]
drawing line in color : RED
drawing line in color : BLUE
```

↥ back to top

Q. What is the difference between fail-fast and fail-safe iterator?

fail-fast Iterator

`Iterators` in java are used to iterate over the Collection objects.Fail-Fast iterators immediately throw `ConcurrentModificationException` if there is **structural modification** of the collection. Structural modification means adding, removing or updating any element from collection while a thread is iterating over that collection. Iterator on ArrayList, HashMap classes are some examples of fail-fast Iterator.

```
import java.util.ArrayList;
import java.util.Iterator;

public class FailFastIteratorExample
{
    public static void main(String[] args) {

        //Creating an ArrayList of integers
        ArrayList list = new ArrayList();

        //Adding elements to list
        list.add(1452);
        list.add(6854);
        list.add(8741);

        //Getting an Iterator from list
        Iterator it = list.iterator();

        while (it.hasNext()) {
            Integer integer = (Integer) it.next();
            list.add(8457);      //This will throw ConcurrentModificationException
        }
    }
}
```

Output

```
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.ArrayList$Itr.checkForComodification(Unknown Source)
    at java.util.ArrayList$Itr.next(Unknown Source)
    at pack1.MainClass.main(MainClass.java:32)
```

fail-safe Iterator

Fail-Safe iterators don't throw any exceptions if a collection is structurally modified while iterating over it. This is because, they operate on the clone of the collection, not on the original collection and that's why they are called fail-safe iterators. Iterator on CopyOnWriteArrayList, ConcurrentHashMap classes are examples of fail-safe Iterator.

```
import java.util.Iterator;
import java.util.concurrent.ConcurrentHashMap;

public class FailSafeIteratorExample
{
    public static void main(String[] args) {

        //Creating a ConcurrentHashMap
        ConcurrentHashMap map = new ConcurrentHashMap();

        //Adding elements to map
        map.put("ONE", 1);
        map.put("TWO", 2);
        map.put("THREE", 3);

        //Getting an Iterator from map
        Iterator it = map.keySet().iterator();

        while (it.hasNext()) {
            String key = (String) it.next();
            System.out.println(key+" : "+map.get(key));
            map.put("FOUR", 4); //This will not be reflected in the Iterator
        }
    }
}
```

Output

```
TWO : 2
FOUR : 4
```

```
ONE : 1
THREE : 3
```

↥ back to top

Q. What are concurrent collection classes?

The concurrent collection APIs of Java provide a range of classes that are specifically designed to deal with concurrent operations. These classes are alternatives to the Java Collection Framework and provide similar functionality except with the additional support of concurrency.

Java Concurrent Collection Classes

- BlockingQueue
- ArrayBlockingQueue
- SynchronousQueue
- PriorityBlockingQueue
- LinkedBlockingQueue
- DelayQueue
- BlockingDeque
- LinkedBlockingDeque
- TransferQueue
- LinkedTransferQueue
- ConcurrentMap
- ConcurrentHashMap
- ConcurrentNavigableMap
- ConcurrentSkipListMap

Q. What is BlockingQueue? How to implement producer-consumer problem by using BlockingQueue?

BlockingQueue: When a thread try to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. Also, when a thread try to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.

Producter-Consumer Problem

Producer and Consumer are two separate threads which share a same bounded Queue. The role of producer to produce elements and push to the queue. The producer halts producing if the queue is full and resumes producing when the size of queue is not full. The consumer consumes the element from the queue. The consumers halt consuming if the size of queue is 0 (empty) and resumes consuming once the queue has an element.

The problem can be approached using various techniques

- Using wait() and notifyAll()
- Using BlockingQueue
- Using sempahores


```
public class ProducerConsumerBlockingQueue {

  static int MAX_SIZE = 5;
  static BlockingQueue queue = new LinkedBlockingQueue(MAX_SIZE);

  public static void main(String[] args) {

    Producer producer = new Producer();
    Consumer consumer = new Consumer();
    producer.start();
    consumer.start();
  }

  static class Producer extends Thread {
    Random random = new Random();

    public void run() {
```

```
      while (true) {
        int element = random.nextInt(MAX_SIZE);
        try {
          queue.put(element);
        } catch (InterruptedException e) {
        }
      }
    }
  }

  static class Consumer extends Thread {
    public void run() {
      while (true) {
        try {
          System.out.println("Consumed " + queue.take());
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    }
  }
}
```

Output

```
Producer 2
Producer 3
Consumed 2
Consumed 3
Producer 0
Producer 4
Consumed 0
```

Here, The Producer start producing objects and pushing it to the Queue. Once the queue is full, the producer will wait until consumer consumes it and it will start producing again. Similar behavior is displayed by consumer. where the consumer waits until there is a single element in queue. It will resume consumer once the queue has element.

↥ back to top

Q. What is difference between Enumeration and Iterator interface?

Enumeration and Iterator are two interfaces in java.util package which are used to traverse over the elements of a Collection object.

Differences

Iterator	Enumeration
hasNext()	hasMoreElements()
next()	nextElement()
remove()	(Not Available)

Sl.No	Enumeration	Iterator
01.	Using Enumeration, you can only traverse the collection. You can't do any modifications to collection while traversing it.	Using Iterator, you can remove an element of the collection while traversing it.
02.	Enumeration is introduced in JDK 1.0	Iterator is introduced from JDK 1.2
03.	Enumeration is used to traverse the legacy classes like Vector, Stack and HashTable.	Iterator is used to iterate most of the classes in the collection framework like ArrayList, HashSet, HashMap, LinkedList etc.
04.	Methods : hasMoreElements() and nextElement()	Methods : hasNext(), next() and remove()
05.	Enumeration is fail-safe in nature.	Iterator is fail-fast in nature.
06.	Enumeration is not safe and secured due to it's fail-safe nature.	Iterator is safer and secured than Enumeration.

```
import java.util.*;
public class PerformanceTest {
```

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

        Vector v = new Vector();
        Object element;
        Enumeration enum;
        Iterator iter;
        long start;

        for(int i = 0; i < 1000000; i++) {
            v.add("New Element");
        }

        enum = v.elements();
        iter = v.iterator();
        // ITERATOR
        start = System.currentTimeMillis();
        while(iter.hasNext()) {
            element = iter.next();
        }
        System.out.println("Iterator took " + (System.currentTimeMillis() - start));
        System.gc();   //request to GC to free up some memory

        // ENUMERATION
        start = System.currentTimeMillis();
        while(enum.hasMoreElements()) {
            element = enum.nextElement();
        }
        System.out.println("Enumeration took " + (System.currentTimeMillis() - start));
    }
}
```

↥ back to top

Q. What is difference between Iterator and ListIterator?

ListIterator is the child interface of Iterator interface. The major difference between Iterator and ListIterator is that Iterator can traverse the elements in the collection only in **forward direction** whereas, the ListIterator can traverse the elements in a collection in both the **forward as well as the backwards direction**.

```
import java.io.*;
import java.util.*;

class IteratorExample
{
 public static void main(String[] args) {

   ArrayList list = new ArrayList();

   list.add(10);
   list.add(20);
   list.add(30);
   list.add(40);
   list.add(50);

   // Iterator
   Iterator itr = list.iterator();
   System.out.println("Iterator:");
   System.out.println("Forward traversal: ");

   while (itr.hasNext())
    System.out.print(itr.next() + " ");

   // ListIterator
   ListIterator i = list.listIterator();
   System.out.println("\nListIterator:");
   System.out.println("Forward Traversal : ");

   while (i.hasNext())
    System.out.print(i.next() + " ");

   System.out.println("\nBackward Traversal : ");

   while (i.hasPrevious())
    System.out.print(i.previous() + " ");
 }
}
```

Output

```
Iterator:
Forward traversal:
10 20 30 40 50

ListIterator:
Forward Traversal :
10 20 30 40 50

Backward Traversal :
50 40 30 20 10
```

Methods of Iterator Interface

No.	Method	Description
1	public boolean hasNext()	It returns true if iterator has more elements.
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 rarely used.

↥ back to top

Q. How can we create a synchronized collection from given collection?

In Java, normally collections aren't synchronized, which leads to fast performance. However, in multi-threaded situations, it can be very useful for collections to be synchronized. The Java Collections class has several static methods on it that provide synchronized collections. These methods are:

- Synchronized Collection Methods of Collections class
- Collections.synchronizedCollection(Collection c)
- Collections.synchronizedList(List list)
- Collections.synchronizedMap(Map<K,V> m)
- Collections.synchronizedSet(Set s)
- Collections.synchronizedSortedMap(SortedMap<K,V> m)
- Collections.synchronizedSortedSet(SortedSet s)

```
/**
* Java program to demonstrate synchronizedCollection()
*
**/
import java.util.*;

public class synchronizedCollectionExample
{

 public static void main(String[] argv) {

  try {

   // creating object of List
   List list = new ArrayList();

   // populate the list
   list.add(10);
   list.add(20);
   list.add(30);
   list.add(40);
   list.add(50);

   // printing the Collection
   System.out.println("Collection : " + list);

   // getting the syncronised view of Collection
   Collection c = Collections.synchronizedCollection(list);

   // printing the Collection
   System.out.println("Synchronized view is: " + c);

  } catch (IllegalArgumentException e) {
   System.out.println("Exception thrown: " + e);
```

```
    }
  }
}
```

Output

```
Collection : [10, 20, 30, 40, 50]
Synchronized view is : [10, 20, 30, 40, 50]
```

↥ back to top

Q. What is a default capacity of ArrayList, Vector, HashMap, Hashtable and Hashset?

Collections	Capacity
ArrayList	10
Vector	10
HashSet	16
HashMap	16
HashTable	11
HashSet	16

- **ArrayList**: Constructs an empty list with an initial capacity of 10.
- **Vector**: Constructs an empty vector so that its internal data array has size 10 and its standard capacity increment is zero.
- **HashMap**: Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).
- **Hashtable**: Constructs a new, empty hashtable with a default initial capacity (11) and load factor (0.75).
- **Hashset**: Constructs a new, empty set; the backing HashMap instance has default initial capacity (16) and load factor (0.75).

Q. What is the difference between Collection and Collections?

Collection Interface

Collection is a root level interface of the Java Collection Framework. Most of the classes in Java Collection Framework inherit from this interface. **List, Set and Queue** are main sub interfaces of this interface. JDK provides direct implementations of it's sub interfaces. **ArrayList, Vector, HashSet, LinkedHashSet, PriorityQueue** are some indirect implementations of Collection interface.

Collections Class

Collections is an utility class in java.util package. It consists of only static methods which are used to operate on objects of type Collection.

Collections Methods	Description
Collections.max()	This method returns maximum element in the specified collection.
Collections.min()	This method returns minimum element in the given collection.
Collections.sort()	This method sorts the specified collection.
Collections.shuffle()	This method randomly shuffles the elements in the specified collection.
Collections.synchronizedCollection()	This method returns synchronized collection backed by the specified collection.
Collections.binarySearch()	This method searches the specified collection for the specified object using binary search algorithm.
Collections.disjoint()	This method returns true if two specified collections have no elements in c
Collections.copy()	This method copies all elements from one collection to another collectio
Collections.reverse()	This method reverses the order of elements in the specified collection.

Q. What is the difference between HashSet and TreeSet?

1. HashSet gives better performance (faster) than TreeSet for the operations like add, remove, contains, size etc. HashSet offers constant time cost while TreeSet offers log(n) time cost for such operations.

2. HashSet does not maintain any order of elements while TreeSet elements are sorted in ascending order by default.

```
import java.util.HashSet;
class HashSetExample {

  public static void main(String[] args) {
     // Create a HashSet
     HashSet hset = new HashSet();

     //add elements to HashSet
     hset.add("Abhijeet");
     hset.add("Ram");
     hset.add("Kevin");
     hset.add("Singh");
     hset.add("Rick");
     // Duplicate removed
     hset.add("Ram");

     // Displaying HashSet elements
     System.out.println("HashSet contains: ");
     for(String temp : hset){
        System.out.println(temp);
     }
  }
}
```

Output

```
HashSet contains:

Rick
Singh
Ram
Kevin
Abhijeet
```

```
import java.util.TreeSet;
class TreeSetExample {

  public static void main(String[] args) {
     // Create a TreeSet
     TreeSet tset = new TreeSet();

     //add elements to TreeSet
     tset.add("Abhijeet");
     tset.add("Ram");
     tset.add("Kevin");
     tset.add("Singh");
     tset.add("Rick");
     // Duplicate removed
     tset.add("Ram");

     // Displaying TreeSet elements
     System.out.println("TreeSet contains: ");
     for(String temp : tset){
        System.out.println(temp);
     }
  }
}
```

Output: Elements are sorted in ascending order.

```
TreeSet contains:

Abhijeet
Kevin
Ram
Rick
Singh
```


↥ back to top

Q. What is the difference between Set and Map?

Sets:-

1. Set does not allow duplicates. Set and all of the classes which implements Set interface should have unique elements.
2. Set allows single null value at most.
3. Set does not maintain any order; still few of its classes sort the elements in an order such as LinkedHashSet maintains the elements in insertion order.
4. Classes used in sets are Set: HashSet, Linked HashSet, TreeSet, SortedSet etc.

Maps:-

1. Map stored the elements as key & value pair. Map doesn't allow duplicate keys while it allows duplicate values.
2. Map can have single null key at most and any number of null values.
3. Set Map also doesn't stores the elements in an order, however few of its classes does the same.
4. Classes in Maps HashMap, TreeMap, WeakHashMap, LinkedHashMap, IdentityHashMap etc.

Q. What is the difference between HashSet and HashMap?

HashSet:-

1. HashSet class implements the Set interface
2. In HashSet, we store objects(elements or values) e.g. If we have a HashSet of string elements then it could depict a set of HashSet elements: {"Hello", "Hi", "Bye", "Run"}
3. HashSet does not allow duplicate elements that mean you can not store duplicate values in HashSet.
4. HashSet permits to have a single null value.
5. HashSet is not synchronized which means they are not suitable for thread-safe operations until unless synchronized explicitly.

HashMap:-

1. HashMap class implements the Map interface
2. HashMap is used for storing key & value pairs. In short, it maintains the mapping of key & value (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This is how you could represent HashMap elements if it has integer key and value of String type: e.g. {1->"Hello", 2->"Hi", 3->"Bye", 4->"Run"}
3. HashMap does not allow duplicate keys however it allows having duplicate values.
4. HashMap permits single null key and any number of null values.
5. HashMap is not synchronized which means they are not suitable for thread-safe operations until unless synchronized explicitly.

Q. What is the difference between HashMap and TreeMap?

Java **HashMap** and **TreeMap** both are the classes of the Java Collections framework. Java Map implementation usually acts as a bucketed hash table. When buckets get too large, they get transformed into nodes of **TreeNodes**, each structured similarly to those in java.util.TreeMap.

HashMap	TreeMap
Java HashMap is a hashtable based implementation of Map interface.	Java TreeMap is a Tree structure-based implementation of Map interface.
HashMap implements Map, Cloneable, and Serializable interface.	TreeMap implements NavigableMap, Cloneable, and Serializable interface.
HashMap allows a **single** null key and multiple null values.	TreeMap does not allow **null** keys but can have multiple null values.
HashMap allows heterogeneous elements because it does not perform sorting on keys.	TreeMap allows homogeneous values as a key because of sorting.
HashMap is **faster** than TreeMap because it provides constant-time performance that is O(1) for the basic operations like get() and put().	TreeMap is **slow** in comparison to HashMap because it provides the performance of O(log(n)) for most operations like add(), remove() and contains().

HashMap	TreeMap
The HashMap class uses the **hash table**.	TreeMap internally uses a **Red-Black** tree, which is a self-balancing Binary Search Tree.
It uses **equals()** method of the Object class to compare keys. The equals() method of Map class overrides it.	It uses the **compareTo()** method to compare keys.
HashMap class contains only basic functions like get(), put(), KeySet(), etc. .	TreeMap class is rich in functionality, because it contains functions like: tailMap(), firstKey(), lastKey(), pollFirstEntry(), pollLastEntry().
Order of elements HashMap does not maintain any order.	The elements are sorted in natural order (ascending).
The HashMap should be used when we do not require key-value pair in sorted order.	The TreeMap should be used when we require key-value pair in sorted (ascending) order.

↥ back to top

Q. What is the Dictionary class?

util.Dictionary is an abstract class, representing a key-value relation and works similiar to a map. Both keys and values can be objects of any type but not null. An attempt to insert either a null key or a null value to a dictionary causes a NullPointerException exception.

```
/**
* Dictionary class Example using
* put(), elements(), get(), isEmpty(), keys() remove(), size()
* Methods
*
**/
import java.util.*;
public class DictionaryExample
{
 public static void main(String[] args) {

   // Initializing a Dictionary
   Dictionary dictionary = new Hashtable();

   // put() method
   dictionary.put("10", "Code");
   dictionary.put("20", "Program");

   // elements() method :
   for (Enumeration i = dictionary.elements(); i.hasMoreElements();) {
    System.out.println("Value in Dictionary : " + i.nextElement());
   }

   // get() method :
   System.out.println("\nValue at key = 6 : " + dictionary.get("6"));
   System.out.println("Value at key = 20 : " + dictionary.get("10"));

   // isEmpty() method :
   System.out.println("\nThere is no key-value pair : " + dictionary.isEmpty() + "\n");

   // keys() method :
   for (Enumeration k = dictionary.keys(); k.hasMoreElements();) {
    System.out.println("Keys in Dictionary : " + k.nextElement());
   }

   // remove() method :
   System.out.println("\nRemove : " + dictionary.remove("10"));
   System.out.println("Check the value of removed key : " + dictionary.get("10"));
   System.out.println("\nSize of Dictionary : " + dictionary.size());
 }
}
```

Output

```
Value in Dictionary : Code
Value in Dictionary : Program

Value at key = 6 : null
Value at key = 20 : Code

There is no key-value pair : false
```

```
Keys in Dictionary : 10
Keys in Dictionary : 20

Remove : Code
Check the value of removed key : null

Size of Dictionary : 1
```

↥ back to top

Q. What are all the Classes and Interfaces that are available in the collections?

Java Collections Interfaces

- Collection Interface
- Iterator Interface
- Set Interface
- List Interface
- Queue Interface
- Dequeue Interface
- Map Interface
- ListIterator Interface
- SortedSet Interface
- SortedMap Interface

Java Collections Classes

- HashSet Class
- TreeSet Class
- ArrayList Class
- LinkedList Class
- HashMap Class
- TreeMap Class
- PriorityQueue Class

Q. What is the difference between HashMap and ConcurrentHashMap?

HashMap	ConcurrentHashMap
HashMap is not synchronized.	ConcurrentHashMap is synchronized.
HashMap is not thread safe.	ConcurrentHashMap is thread safe.
HashMap iterator is fail-fast and ArrayList throws ConcurrentModificationException if concurrent modification happens during iteration.	ConcurrentHashMap is fail-safe and it will never throw ConcurrentModificationException during iteration.
HashMap allows key and value to be null.	ConcurrentHashMap does not allow null key/value. It will throw NullPointerException.
HashMap is faster.	ConcurrentHashMap is slower than HashMap.

Q. What is CopyOnWriteArrayList? How it is different from ArrayList in Java?

CopyOnWriteArrayList class is introduced in JDK 1.5, which implements List interface. It is enhanced version of ArrayList in which all modifications (add, set, remove, etc) are implemented by making a fresh copy.

```
/**
* Java program to illustrate
* CopyOnWriteArrayList class
*
**/
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.*;

class ConcurrentDemo extends Thread {
```

```
 static CopyOnWriteArrayList arrList = new CopyOnWriteArrayList();

 public void run() {
  // Child thread trying to
  // add new element in the
  // Collection object
  arrList.add("D");
 }

 public static void main(String[] args)
  throws InterruptedException {
  arrList.add("A");
  arrList.add("B");
  arrList.add("c");

  // We create a child thread
  // that is going to modify
  // ArrayList.
  ConcurrentDemo t = new ConcurrentDemo();
  t.run();

  Thread.sleep(1000);

  // Now we iterate through
  // the ArrayList and get
  // exception.
  Iterator itr = arrList.iterator();
  while (itr.hasNext()) {
   String s = (String)itr.next();
   System.out.println(s);
   Thread.sleep(1000);
  }
  System.out.println(arrList);
 }
}
```

Output

```
A
B
c
D
[A, B, c, D]
```

↥ back to top

Q. How to make an ArrayList read only in Java?

An ArrayList can be made read-only easily with the help of **Collections.unmodifiableList()** method. This method takes the modifiable ArrayList as a parameter and returns the read-only unmodifiable view of this ArrayList.

```
/**
* Java program to demonstrate
* unmodifiableList() method
*
**/

import java.util.*;

public class ReadOnlyArrayListExample
{
 public static void main(String[] argv)
  throws Exception {

  try {

   // creating object of ArrayList
   List list = new ArrayList();

   // populate the list
   list.add('X');
   list.add('Y');
   list.add('Z');

   // printing the list
```

```
    System.out.println("Initial list: "+ list);

    // getting readonly list using unmodifiableList() method
    List immutablelist = Collections.unmodifiableList(list);

    // printing the list
    System.out.println("ReadOnly ArrayList: "+ immutablelist);

    // Adding element to new Collection
    System.out.println("\nTrying to modify the ReadOnly ArrayList.");
    immutablelist.add('A');
   } catch (UnsupportedOperationException e) {
    System.out.println("Exception thrown : " + e);
   }
  }
}
```

Output

```
Initial list: [X, Y, Z]
ReadOnly ArrayList: [X, Y, Z]

Trying to modify the ReadOnly ArrayList.
Exception thrown : java.lang.UnsupportedOperationException
```

↥ back to top

Q. Why Collection doesn't extend Cloneable and Serializable interfaces?

Collection is an interface that specifies a group of objects known as elements. The details of how the group of elements is maintained is left up to the concrete implementations of `Collection`. For example, some Collection implementations like `List` allow duplicate elements whereas other implementations like `Set` don't.

Collection is the root interface for all the collection classes (like ArrayList, LinkedList). If collection interface extends Cloneable/Serializable interfaces, then it is mandating all the concrete implementations of this interface to implement cloneable and serializable interfaces. To give freedom to concrete implementation classes, Collection interface don't extended Cloneable or Serializable interfaces.

Q. Why ConcurrentHashMap is faster than Hashtable in Java?

ConcurrentHashMap uses multiple buckets to store data. This avoids read locks and greatly improves performance over a HashTable. Both are thread safe, but there are obvious performance wins with ConcurrentHashMap.

When we read from a ConcurrentHashMap using get(), there are no locks, contrary to the HashTable for which all operations are simply synchronized. HashTable was released in old versions of Java whereas ConcurrentHashMap is added in java 1.5 version.

Q. What is the difference between peek(), poll() and remove() method of the Queue interface?

This represents a collection that is indented to hold data before processing. It is an arrangement of the type First-In-First-Out (FIFO). The first element put in the queue is the first element taken out from it.

The peek() method

This method returns the object at the top of the current queue, without removing it. If the queue is empty this method returns null.

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Queue;
public class QueueExample
{
   public static void main(String args[]) {

      Queue queue = new LinkedList();
```

```
      queue.add("Java");
      queue.add("JavaFX");
      queue.add("OpenCV");
      queue.add("Coffee Script");
      queue.add("HBase");

      System.out.println("Element at the top of the queue: "+queue.peek());
      Iterator it = queue.iterator();
      System.out.println("Contents of the queue: ");
      while(it.hasNext()) {
         System.out.println(it.next());
      }
   }
}
```

Output

```
Element at the top of the queue: Java
Contents of the queue:
Java
JavaFX
OpenCV
Coffee Script
Hbase
```

The poll() method

The poll() method of the Queue interface returns the object at the top of the current queue and removes it. If the queue is empty this method returns null.

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Queue;
public class QueueExample
{
   public static void main(String args[]) {

      Queue queue = new LinkedList();
      queue.add("Java");
      queue.add("JavaFX");
      queue.add("OpenCV");
      queue.add("Coffee Script");
      queue.add("HBase");

      System.out.println("Element at the top of the queue: "+queue.poll());
      Iterator it = queue.iterator();
      System.out.println("Contents of the queue: ");
      while(it.hasNext()) {
         System.out.println(it.next());
      }
   }
}
```

Output

```
Element at the top of the queue: Java
Contents of the queue:
JavaFX
OpenCV
Coffee Script
HBase
```

Differences

Both **poll()** and **remove()** method is used to remove head object of the Queue.

The main difference lies when the Queue is empty(). If Queue is empty then poll() method will return **null**. While in similar case, remove() method will throw **NoSuchElementException**. peek() method retrieves but does not remove the head of the Queue. If queue is empty then peek() method also returns null.

↥ back to top

Q. How HashMap works in Java?

HashMap in Java works on **hashing** principle. It is a data structure which allow to store object and retrieve it in constant time O(1). In hashing, hash functions are used to link key and value in HashMap. Objects are stored by calling **put(key, value)** method of HashMap and retrieved by calling **get(key)** method. When we call put method, **hashcode()** method of the key object is called so that hash function of the map can find a bucket location to store value object, which is actually an index of the internal array, known as the table. HashMap internally stores mapping in the form of **Map.Entry** object which contains both key and value object.

Since the internal array of HashMap is of fixed size, and if you keep storing objects, at some point of time hash function will return same bucket location for two different keys, this is called **collision** in HashMap. In this case, a linked list is formed at that bucket location and a new entry is stored as next node.

If we try to retrieve an object from this linked list, we need an extra check to search correct value, this is done by **equals()** method. Since each node contains an entry, HashMap keeps comparing entry's key object with the passed key using equals() and when it return true, Map returns the corresponding value.

Example:

```
/**
* Java program to illustrate internal working of HashMap
*
**/
import java.util.HashMap;

class Key {
    String key;
    Key(String key) {
        this.key = key;
    }

    @Override
    public int hashCode() {
        int hash = (int)key.charAt(0);
        System.out.println("hashCode for key: "
                           + key + " = " + hash);
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        return key.equals(((Key)obj).key);
    }
}

public class HashMapExample {
    public static void main(String[] args) {
        HashMap map = new HashMap();
        map.put(new Key("Hello"), 20);
        map.put(new Key("World"), 30);
        map.put(new Key("Java"), 40);

        System.out.println();
        System.out.println("Value for key World: " + map.get(new Key("World"))); //hashCode for key: World = 118
        System.out.println("Value for key Java: " + map.get(new Key("Java")));   //hashCode for key: Java = 115
    }
}
```

Q. How does HashMap handle collisions in java?

Prior to Java 8, HashMap and all other hash table based Map implementation classes in Java handle collision by chaining, i.e. they use linked list to store map entries which ended in the same bucket due to a collision. If a key end up in same bucket location where an entry is already stored then this entry is just added at the head of the linked list there. In the worst case this degrades the performance of the `get()` method of HashMap to `O(n)` from `O(1)`. In order to address this issue in the case of frequent HashMap collisions, Java 8 has started using a **balanced tree** instead of linked list for storing collided entries. This also means that in the worst case you will get a performance boost from `O(n)` to `O(log n)`.

The threshold of switching to the balanced tree is defined as TREEIFY_THRESHOLD constant in java.util.HashMap JDK 8 code. Currently, it's value is 8, which means if there are more than 8 elements in the same bucket than HashMap will use a tree instead of linked list to hold them in the same bucket.

Q. Write a code to convert HashMap to ArrayList.

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.Map.Entry;
import java.util.Set;
public class MapToListExamples {

    public static void main(String[] args) {

        // Creating a HashMap object
        HashMap performanceMap = new HashMap();

        // Adding elements to HashMap
        performanceMap.put("John Kevin", "Average");
        performanceMap.put("Ladarious Fernandez", "Very Good");
        performanceMap.put("Ivan Jose", "Very Bad");
        performanceMap.put("Smith Jacob", "Very Good");
        performanceMap.put("Athena Stiltner", "Bad");

        // Getting Set of keys
        Set keySet = performanceMap.keySet();

        // Creating an ArrayList of keys
        ArrayList listOfKeys = new ArrayList(keySet);

        System.out.println("ArrayList Of Keys :");

        for (String key : listOfKeys) {
            System.out.println(key);
        }

        // Getting Collection of values
        Collection values = performanceMap.values();

        // Creating an ArrayList of values
        ArrayList listOfValues = new ArrayList(values);

        System.out.println("ArrayList Of Values :");

        for (String value : listOfValues) {
            System.out.println(value);
        }

        // Getting the Set of entries
        Set> entrySet = performanceMap.entrySet();

        // Creating an ArrayList Of Entry objects
        ArrayList> listOfEntry = new ArrayList>(entrySet);

        System.out.println("ArrayList of Key-Values :");

        for (Entry entry : listOfEntry) {
            System.out.println(entry.getKey()+" : "+entry.getValue());
        }
    }
}
```


↥ back to top

Q. What is difference between arrayList and linkedList?

ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.

ArrayList	LinkedList
ArrayList internally uses a dynamic array to store the elements.	LinkedList internally uses a doubly linked list to store the elements.

ArrayList	LinkedList
Manipulation with ArrayList is slow because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
An ArrayList class can act as a list only because it implements List only.	LinkedList class can act as a list and queue
ArrayList is better for storing and accessing data.	LinkedList is better for manipulating data.

Q. How Set/HashSet implement unique values?

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

- HashSet stores the elements by using a mechanism called hashing.
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16, and the load factor is 0.75.

Example:

```
import java.util.*;
class HashSetExample {

 public static void main(String args[]){

    // Creating HashSet and adding elements
    HashSet set=new HashSet();
           set.add("10");
           set.add("20");
           set.add("30");
           set.add("40");
           set.add("50");
           Iterator i=set.iterator();
           while(i.hasNext()) {
             System.out.println(i.next());
           }
    }
}
```

When we create a HashSet, it internally creates a HashMap and if we insert an element into this HashSet using add() method, it actually call put() method on internally created HashMap object with element you have specified as it's key and constant Object called **PRESENT** as it's value. So we can say that a Set achieves uniqueness internally through HashMap.

↥ back to top

Q. What is Comparable and Comparator Interface in java?

Comparable and Comparator both are interfaces and can be used to sort collection elements.

Comparable	Comparator
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.	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) Comparable affects the original class, i.e., the actual class is modified.	Comparator doesn't affect the original class, i.e., the actual class is not modified.
3) Comparable provides compareTo() method to sort elements.	Comparator provides compare() method to sort elements.
4) Comparable is present in java.lang package.	A Comparator is present in the java.util package.

5. We can sort the list elements of Comparable type by Collections.sort(List) method.|We can sort the list elements of Comparator type by Collections.sort(List, Comparator) method.|

Example:

```
/**
* Java Program to demonstrate the use of Java Comparable.
*
**/
import java.util.*;
import java.io.*;

class Student implements Comparable{
    int rollno;
    String name;
    int age;
    Student(int rollno,String name,int age){
        this.rollno=rollno;
        this.name=name;
        this.age=age;
    }
    public int compareTo(Student st){
        if(age==st.age)
          return 0;
        else if(age>st.age)
          return 1;
        else
          return -1;
    }
}

// Creating a test class to sort the elements
public class ComparableMain {
    public static void main(String args[]) {
        ArrayList al=new ArrayList();
        al.add(new Student(101,"Ryan Frey",23));
        al.add(new Student(106,"Kenna Bean",27));
        al.add(new Student(105,"Jontavius Herrell",21));

        Collections.sort(al);
        for(Student st:al){
            System.out.println(st.rollno+" "+st.name+" "+st.age);
        }
    }
}
```

Example: Java Comparator Student.java

```
class Student {
    int rollno;
    String name;
    int age;
    Student(int rollno,String name,int age) {
      this.rollno=rollno;
      this.name=name;
      this.age=age;
    }
}
```

AgeComparator.java

```
import java.util.*;

class AgeComparator implements Comparator {
    public int compare(Student s1,Student s2) {
    if(s1.age==s2.age)
      return 0;
    else if(s1.age>s2.age)
      return 1;
    else
      return -1;
    }
}
```

NameComparator.java

```
import java.util.*;

class NameComparator implements Comparator {
    public int compare(Student s1,Student s2) {
        return s1.name.compareTo(s2.name);
    }
}
```

TestComparator.java

```
/**
* Java Program to demonstrate the use of Java Comparator
*
**/
import java.util.*;
import java.io.*;

class TestComparator {

    public static void main(String args[]) {
        // Creating a list of students
        ArrayList al=new ArrayList();
        al.add(new Student(101,"Caelyn Romero",23));
        al.add(new Student(106,"Olivea Gold",27));
        al.add(new Student(105,"Courtlyn Kilgore",21));

        System.out.println("Sorting by Name");
        // Using NameComparator to sort the elements
        Collections.sort(al,new NameComparator());
        // Traversing the elements of list
        for(Student st: al){
          System.out.println(st.rollno+" "+st.name+" "+st.age);
        }

        System.out.println("sorting by Age");
        // Using AgeComparator to sort the elements
        Collections.sort(al,new AgeComparator());
        // Travering the list again
        for(Student st: al){
          System.out.println(st.rollno+" "+st.name+" "+st.age);
        }
    }
}
```

Output:

```
Sorting by Name
106 Caelyn Romero 23
105 Courtlyn Kilgore 21
101 Olivea Gold 27

Sorting by Age
105 Courtlyn Kilgore 21
101 Caelyn Romero 23
106 Olivea Gold 27
```

↥ back to top

Q. Difference between containsKey(), keySet() and values() in HashMap.

- **The keySet() method**: This method returns a Set view of all the keys in the map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.
- **The containsKey() method**: It returns true if this map maps one or more keys to the specified value.
- **The values() methods**: It returns a Collection view of the values contained in this map. The collection is backed by the map, so changes to the map are reflected in the collection, and vice-versa.

Example:

```
/**
* Java program illustrating usage of HashMap class methods
```

```
* keySet(), values(), containsKey()
*
**/
import java.util.*;
public class HashMapExample {

    public static void main(String args[]) {

        // Creation of HashMap
        HashMap map = new HashMap<>();

        // Adding values to HashMap as ("keys", "values")
        map.put("Language", "Java");
        map.put("Platform", "Window");
        map.put("Code", "HashMap");
        map.put("Learn", "More");

        // containsKey() method is to check the presence of a particluar key
        if (map.containsKey("Code"))
            System.out.println("Testing .containsKey : " + map.get("Code"));

        // keySet() method returns all the keys in HashMap
        Set mapKeys = map.keySet();
        System.out.println("Initial keys  : " + mapKeys);


        // values() method return all the values in HashMap
        Collection mapValues = map.values();
        System.out.println("Initial values : " + mapValues);

        // Adding new set of key-value
        map.put("Search", "JavaArticle");

        // Again using .keySet() and .values() methods
        System.out.println("New Keys : " + mapKeys);
        System.out.println("New Values: " + mapValues);
    }
}
```

↥ back to top

Q. What is the difference between Array and ArrayList data-structure?

- **Resizable**: Implementation of array is simple fixed sized array but Implementation of ArrayList is dynamic sized array.
- **Primitives**: Array can contain both primitives and objects but ArrayList can contain only object elements
- **Generics**: We can't use generics along with array but ArrayList allows us to use generics to ensure type safety.
- **Length**: We can use length variable to calculate length of an array but size() method to calculate size of ArrayList.
- **Store**: Array use assignment operator to store elements but ArrayList use add() to insert elements.

Example:

```
/*
* A Java program to demonstrate differences between array
* and ArrayList
*
**/
import java.util.ArrayList;
import java.util.Arrays;

class ArrayExample {

    public static void main(String args[]) {

        /* ........... Normal Array..... */
        // Need to specify the size for array
        int[] arr = new int[3];
        arr[0] = 10;
        arr[1] = 20;
        arr[2] = 30;
        // We cannot add more elements to array arr[]

        /*.....ArrayList.....*/
        // Need not to specify size
        ArrayList arrL = new ArrayList();
        arrL.add(10);
        arrL.add(20);
```

```
        arrL.add(30);
        arrL.add(40);
        // We can add more elements to arrL

        System.out.println(arrL);
        System.out.println(Arrays.toString(arr));
    }
}
```

↥ back to top

Q. Array or ArrayList which one is faster?

- Array is faster

Q. What is difference between HashSet and LinkedHashSet?

A HashSet is unordered and unsorted Set. LinkedHashSet is the ordered version of HashSet. The only difference between HashSet and LinkedHashSet is that LinkedHashSet maintains the **insertion order**. When we iterate through a HashSet, the order is unpredictable while it is predictable in case of LinkedHashSet. The reason why LinkedHashSet maintains insertion order is because the underlying data structure is a doubly-linked list.

Q. What is the difference between HashTable and HashMap?

HashMap	Hashtable
HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code.	Hashtable is **synchronized**. It is thread-safe and can be shared with many threads.
HashMap allows one null key and multiple null values.	Hashtable doesn't allow any null key or value.
HashMap is a new class introduced in JDK 1.2.	Hashtable is a legacy class.
HashMap is fast.	Hashtable is slow.
We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap);	Hashtable is internally synchronized and can't be unsynchronized.
HashMap is traversed by Iterator.	Hashtable is traversed by Enumerator and Iterator.
Iterator in HashMap is fail-fast.	Enumerator in Hashtable is not fail-fast.
HashMap inherits AbstractMap class.	Hashtable inherits Dictionary class.

Example:

```
/**
* A sample Java program to demonstrate HashMap and HashTable
*
**/
import java.util.*;
import java.lang.*;
import java.io.*;

class Example
{
    public static void main(String args[]) {
        // HashTable
        Hashtable ht = new Hashtable();
        ht.put(101,"One");
        ht.put(101,"Two");
        ht.put(102,"Three");
        System.out.println("Hash Table Values");
        for (Map.Entry m:ht.entrySet()) {
            System.out.println(m.getKey() + " " + m.getValue());
        }

        // HashMap
        HashMap hm = new HashMap();
        hm.put(100,"Four");
```

```
        hm.put(104,"Four");  // hash map allows duplicate values
        hm.put(101,"Five");
        System.out.println("Hash Map Values");
        for (Map.Entry m:hm.entrySet()) {
            System.out.println(m.getKey() + " " + m.getValue());
        }
    }
}
```

Output:

```
Hash Table Values
102 Three
101 One

Hash Map Values
100 Four
101 Five
104 Four
```

↥ back to top

Q. What happens when a duplicate key is put into a HashMap?

By definition, the `put` command replaces the previous value associated with the given key in the map (conceptually like an array indexing operation for primitive types).

The map simply drops its reference to the value. If nothing else holds a reference to the object, that object becomes eligible for garbage collection. Additionally, Java returns any previous value associated with the given key (or `null` if none present), so you can determine what was there and maintain a reference if necessary.

Q. What are the differences between ArrayList and Vector?

ArrayList	Vector
ArrayList is **not synchronized**.	Vector is **synchronized**.
ArrayList **increments 50%** of current array size if the number of elements exceeds from its capacity.	Vector **increments 100%** means doubles the array size if the total number of elements exceeds than its capacity.
ArrayList is not a legacy class. It is introduced in JDK 1.2.	Vector is a legacy class.
ArrayList is **fast** because it is non-synchronized.	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.
ArrayList uses the **Iterator** interface to traverse the elements.	A Vector can use the **Iterator** interface or **Enumeration** interface to traverse the elements.

Example:

```
/**
* Java Program to illustrate use of ArrayList
* and Vector in Java
*
**/
import java.io.*;
import java.util.*;

class Example
{
    public static void main (String[] args) {
        // creating an ArrayList
        ArrayList arrlist = new ArrayList();

        // adding object to arraylist
        arrlist.add("One");
        arrlist.add("Two");
        arrlist.add("Three");
```

```
        // traversing elements using Iterator'
        System.out.println("ArrayList elements are:");
        Iterator itr = arrlist.iterator();
        while (itr.hasNext())
            System.out.println(itr.next());

        // creating Vector
        Vector vtr = new Vector();
        vtr.addElement("Four");
        vtr.addElement("Five");
        vtr.addElement("Six");

        // traversing elements using Enumeration
        System.out.println("\nVector elements are:");
        Enumeration eum = vtr.elements();
        while (eum.hasMoreElements())
            System.out.println(eum.nextElement());
    }
}
```

Output:

```
ArrayList elements are:
One
Two
Three

Vector elements are:
Four
Five
Six
```

↥ back to top

Q. If you store Employee object as key say: Employee emp = new Employee(“name1”,20); store it in a HashMap as key, now if we add a new parameter emp.setMarriedStatus(true) and try to override it what will happen?

new instance of Employee will be inserted to HashMap

Q. Why Map interface does not extend Collection interface?

Q. What is CompareAndSwap approach?

↥ back to top