Traditional Collections

Collections

Why do we need Collections in Java?

Arrays are not dynamic. Once an array of a particular size is declared, the size cannot be modified. To add a new element to the array, a new array has to be created with bigger size and all the elements from the old array copied to new array.

Collections are used in situations where data is dynamic. Collections allow adding an element, deleting an element and host of other operations. There are a number of Collections in Java allowing to choose the right Collection for the right context.

What are the important interfaces in the Collection Hierarchy?

The most important interfaces and their relationships are highlighted below.

```
interface Collection<E> extends Iterable<E> {
}

// Unique things only - Does not allow duplication.
// If obj1.equals(obj2) then only one of them can be in the Set.
interface Set<E> extends Collection<E> {

}

// LIST OF THINGS
// Cares about which position each object is in
// Elements can be added in by specifying position - where should it be added
in
// If element is added without specifying position - it is added at the end
interface List<E> extends Collection<E> {
}
```

```
// Arranged in order of processing - A to-do list for example
// Queue interface extends Collection. So, it supports all Collection
Methods.
interface Queue<E> extends Collection<E> {
}
```

```
// A,C,A,C,E,C,M,D,H,A => {("A",5),("C",2)}
// Key - Value Pair {["key1",value1],["key2",value2],["key3",value3]}
// Things with unique identifier
interface Map<K, V> {
}
```

```
interface Collection<E> extends Iterable<E>
{
  boolean add(E paramE);
  boolean remove(Object paramObject);

  int size();
  boolean isEmpty();
  void clear();

  boolean contains(Object paramObject);
  boolean containsAll(Collection<?> paramCollection);

  boolean addAll(Collection<? extends E> paramCollection);
  boolean removeAll(Collection<?> paramCollection);
  boolean retainAll(Collection<?> paramCollection);
```

```
   Iterator<E> iterator();

   //A NUMBER OF OTHER METHODS AS WELL..
 }
```

```
public abstract interface Iterable<T>
{
  public abstract Iterator<T> iterator();
}
```

Note: In particular, `c1.equals(c2)` implies(kaa tatpary) that `c1.hashCode()==c2.hashCode()`.

Can you explain briefly about the List Interface?

List interface extends Collection interface. So, it contains all methods defined in the Collection interface. In addition, List interface allows operation specifying the position of the element in the Collection.

Most important thing to remember about a List interface - any implementation of the List interface would maintain the insertion order. When an element A is inserted into a List (without specifying position) and then another element B is inserted, A is stored before B in the List.
When a new element is inserted without specifying a position, it is inserted at the end of the list of elements.

However, We can also use the void add(int position, E paramE); method to insert an element at a specific position.

Listed below are some of the important methods in the List interface (other than those inherited from Collection interface):

```
interface List<E> extends Collection<E>
{
  boolean addAll(int paramInt, Collection<? extends E> paramCollection);

  E get(int paramInt);

  E set(int paramInt, E paramE);

  void add(int paramInt, E paramE);
  E remove(int paramInt);

  int indexOf(Object paramObject);
  int lastIndexOf(Object paramObject);

  ListIterator<E> listIterator();
  ListIterator<E> listIterator(int paramInt);
  List<E> subList(int paramInt1, int paramInt2);
}
```

```
class ArrayList /* implements List<E>, RandomAccess */{
    // implements RandomAccess, a marker interface, meaning it support fast -
    // almost constant time - access
    // Insertion and Deletion are slower compared to LinkedList
}

class Vector /* implements List<E>, RandomAccess */{
    // Thread Safe - Synchronized Methods
    // implements RandomAccess, a marker interface, meaning it support fast -
    // almost constant time - access
}
```

```
class LinkedList /* implements List<E>, Queue */{
    // Elements are doubly linked - forward and backword - to one another
    // Ideal choice to implement Stack or Queue
    // Iteration is slower than ArrayList
    // Fast Insertion and Deletion
    // Implements Queue interface also. So, supports methods like peek(), poll
    // and remove()
}
```

Set Interface:-

```
// Unique things only - Does not allow duplication.
// If obj1.equals(obj2) then only one of them can be in the Set.
interface Set<E> extends Collection<E> {

}
```

SortedsSet interface maintains order but Set interface does not maintains ordering.

```
//Main difference between Set and SortedSet is - an implementation of SortedSe
//maintains its elements in a sorted order.  Set interface does not guarantee
interface SortedSet<E> extends Set<E> {

    SortedSet<E> subSet(E fromElement, E toElement);

    SortedSet<E> headSet(E toElement);

    SortedSet<E> tailSet(E fromElement);

    E first();

    E last();

}
```

Navigate closest matches for a give search targets.

.

```
interface NavigableSet<E> extends SortedSet<E> {
    E lower(E e);

    E floor(E e);

    E ceiling(E e);

    E higher(E e);

    E pollFirst();

    E pollLast();
}
```

HashSet is implementation of **Set** Interface which does not allow duplicate value all the methods which are in **Collection Framework** are also in Set Interface by default but when we are talking about Hash set the main thing is **objects which are going to be stored in HashSet must override equals() and hashCode() method** so that we can check for equality and no duplicate value are stored in our set.if we have created our own objects we need to implement **hashCode**() and **equal**() in such a manner that will be able to compare objects correctly when

storing in a set so that duplicate objects are not stored,if we have not override this method objects will take default implementation of this method.

public boolean add(Object o) Method is used to add element in a set which returns false if it's a duplicate value in case of HashSet otherwise returns true if added successfully.

```
// Order of Insertion : A, X , B
// Possible Order of Storing : X, A ,B
class HashSet /* implements Set */{
    // unordered, unsorted - iterates in random order
    // uses hashCode()
}

// Order of Insertion :A, X, B
// Order of Storing : A, X, B
class LinkedHashSet /* implements Set */{
    // ordered - iterates in order of insertion
    // unsorted
    // uses hashCode()
}
```

```
// Order of Insertion :A,C,B
// Order of Storing : A,B,C
class TreeSet /* implements Set,NavigableSet */{
    // 3,5,7
    // sorted - natural order
    // implements NavigableSet
}
```

Queue Interface:-

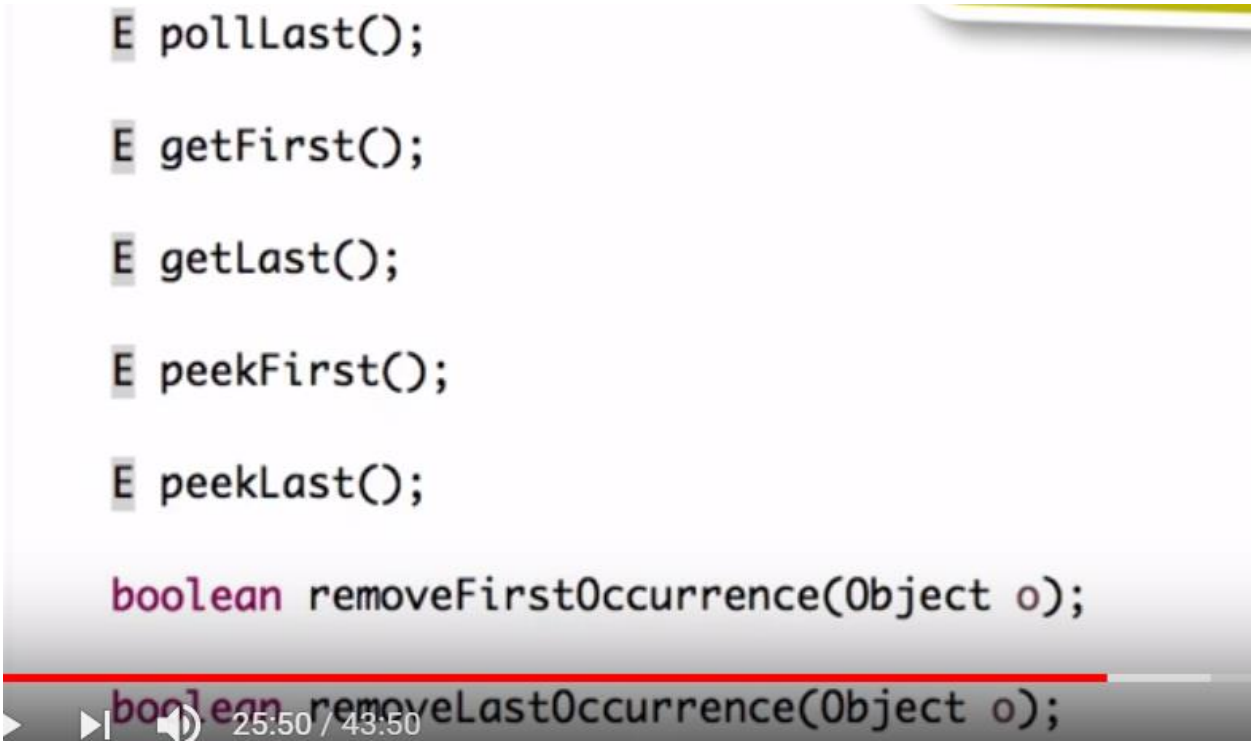
```java
// Arranged in order of processing - A to-do list for example
// Queue interface extends Collection. So, it supports all Collection Methods.
interface Queue<E> extends Collection<E> {

    //Inserts the specified element into this queue
    //Throws exception in case of failure
    boolean add(E paramE);

    //Inserts the specified element into this queue
    //Returns false in case of failure
    boolean offer(E paramE);
```

```java
    //Retrieves and removes the head of this queue.
    //Throws Exception if Queue is empty
    E remove();

    //Retrieves and removes the head of this queue.
    //returns null if Queue is empty
    E poll();
```

```java
    E element();

    E peek();
}
```

```java
//A linear collection that supports e
interface Deque<E> extends Queue<E> {
    void addFirst(E e);

    void addLast(E e);

    boolean offerFirst(E e);

    boolean offerLast(E e);

    E removeFirst();

    E removeLast();

    E pollFirst();
```

Queue with Processing at both ends...

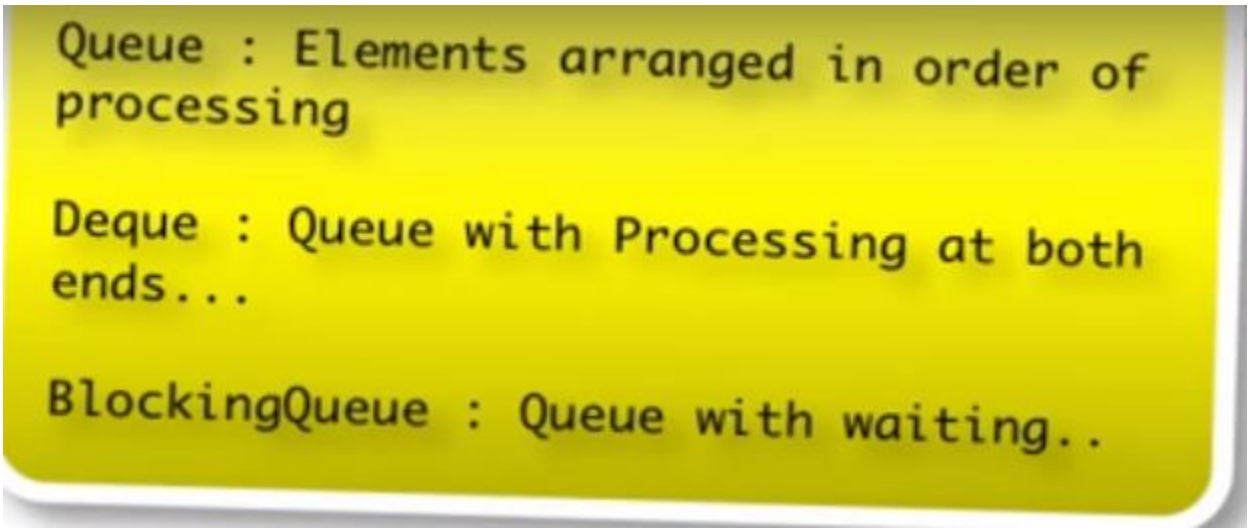


BlockingQueue Interface:-

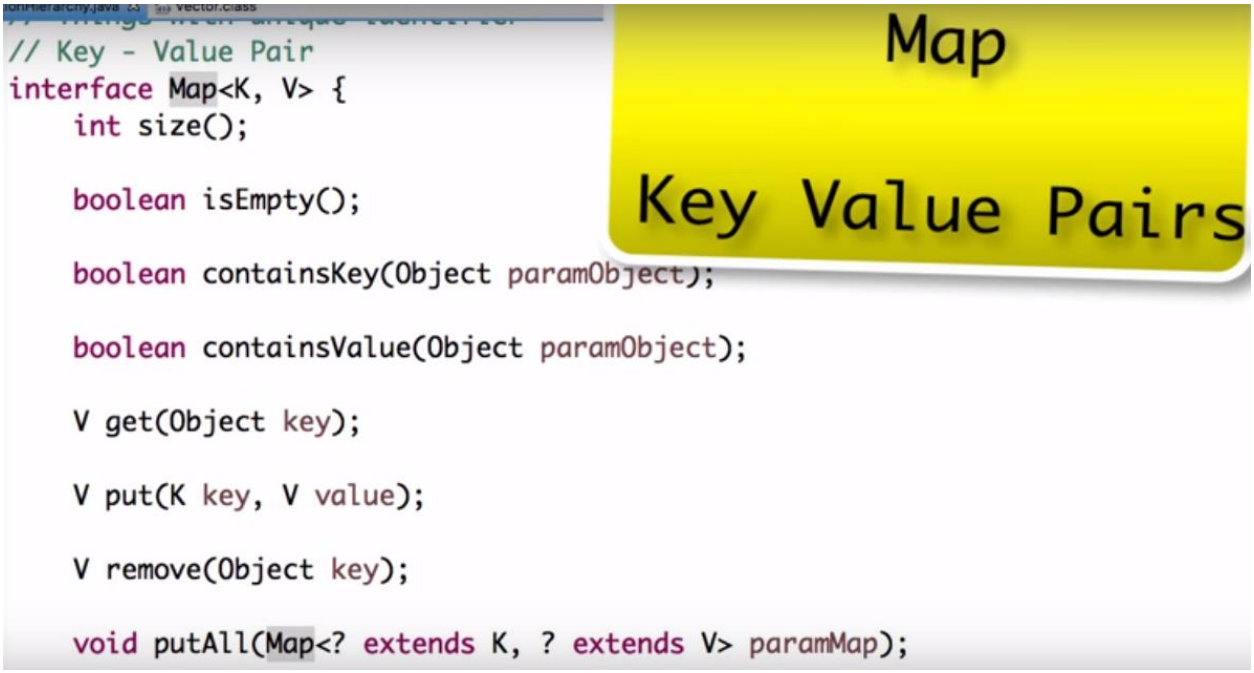
```
//A Queue that additionally supports operations that wait for
//the queue to become non-empty when retrieving an
//element, and wait for space to become available in the queue when
//storing an element.
interface BlockingQueue<E> extends Queue<E> {
    //Same as in Queue Interface
    //Inserts the specified element into queue IMMEDIATELY
    //Throws exception in case of failure
    boolean add(E e);

    //Same as in Queue Interface
    //Inserts the specified element into queue IMMEDIATELY
    //Returns false in case of failure
    boolean offer(E e); //Same as in Queue Interface

    //Inserts the specified element into this queue, waiting
    //if necessary for space to become available.
    void put(E e) throws InterruptedException;

    //waiting up to the specified wait time
    boolean offer(E e, long timeout, TimeUnit unit) throws InterruptedExceptio

    //waits until element becomes available
    E take() throws InterruptedException;

    //waits for specified time and returns null if time expires
    E poll(long timeout, TimeUnit unit) throws InterruptedException;

    int remainingCapacity();

    boolean remove(Object o);

    public boolean contains(Object o);
```

Classes of queue interface:-

```
//The elements of the priority queue are ordered according to their natural or
class PriorityQueue /* implements Queue */{
    // sorted - natural order

}

class ArrayDeque /*implements Deque*/{

}

class ArrayBlockingQueue /*implements BlockingQueue*/{
    //uses Array - optionally-bounded
}

class LinkedBlockingQueue /*implements BlockingQueue*/{
    //uses Linked List - optionally-bounded
    //Linked queues typically have higher throughput than array-based queues b
    //less predictable performance in most concurrent applications.
```



Map Interface:




```java
// Key - Value Pair
interface Map<K, V> {
    int size();

    boolean isEmpty();

    boolean containsKey(Object paramObject);

    boolean containsValue(Object paramObject);

    V get(Object key);

    V put(K key, V value);

    V remove(Object key);

    void putAll(Map<? extends K, ? extends V> paramMap);

    void clear();

    Set<K> keySet();

    Collection<V> values();

    Set<Entry<K, V>> entrySet();

    boolean equals(Object paramObject);

    int hashCode();

    public static abstract interface Entry<K, V> {
        K getKey();

        V getValue();

        V setValue(V paramV);
```

```
        boolean equals(Object paramObject);

        int hashCode();
    }
}
```

```
// A Map that orders based on the keys. Comparator can be provided at
// map creation time
interface SortedMap<K, V> extends Map<K, V> {
    Comparator<? super K> comparator();

    SortedMap<K, V> subMap(K fromKey, K toKey);

    SortedMap<K, V> headMap(K toKey);

    SortedMap<K, V> tailMap(K fromKey);

    K firstKey();

    K lastKey();
}
```

Navigate closest matches for a give search targets.

```
interface NavigableMap<K, V> extends SortedMap<K, V> {
    Map.Entry<K, V> lowerEntry(K key);

    K lowerKey(K key);

    Map.Entry<K, V> floorEntry(K key);

    K floorKey(K key);

    Map.Entry<K, V> ceilingEntry(K key);

    K ceilingKey(K key);

    Map.Entry<K, V> higherEntry(K key);

    K higherKey(K key);
```

Classes for Map Interface:-

```
class HashMap /* implements Map */{
    // unsorted, unordered
    // key's hashCode() is used
}

class Hashtable /* implements Map */{
    // Synchronized - Thread Safe - version of HashMap
    // unsorted, unordered
    // key's hashCode() is used
    // HashMap allows a key with null value. Hashtable doesn't.
}
```

```
}

class LinkedHashMap /* implements Map
    // insertion order is maintained
    // well)
    // slower insertion and deletion
    // faster iteration
}

// A,C,B
// A,B,C
class TreeMap /* implements Map,NavigableMap */{
    // sorted order is maintained
    // implements NavigableMap
}
```

Map and Set have similarly named interfaces and classes.

What are the static methods present in the Collections class?

- static int binarySearch(List, key)
  - Can be used only on sorted list
- static int binarySearch(List, key, Comparator)
- static void reverse(List)
  - Reverse the order of elements in a List.
- static Comparator reverseOrder();
  - Return a Comparator that sorts the reverse of the collection current sort sequence.
- static void sort(List)
- static void sort(List, Comparator)

Q1 What is Collection ? What is a Collections Framework ? What are the benefits of Java Collections Framework ?

Collection : A collection (also called as container) is an object that groups multiple elements into a single unit.

Collections Framework : Collections framework provides unified architecture for manipulating and representing collections.

Benefits of Collections Framework :

1. Improves program quality and speed
2. Increases the chances of reusability of software
3. Decreases programming effort.

Q2 What is the root interface in collection hierarchy ?

Root interface in collection hierarchy is **Collection interface .** Few interviewer may argue that
Collection interface extends **Iterable interface**. So iterable should be the root interface. But you should reply iterable interface present in java.lang package not in java.util package .It is clearly mentioned in Oracle Collection docs , that Collection interface is a member of the Java Collections framework. For Iterable interface Oracle doc , iterable interface is not mentioned as a part of the Java Collections framework .So if the question includes collection hierarchy , then you should answer the question as Collection interface (which is found in java.util package).

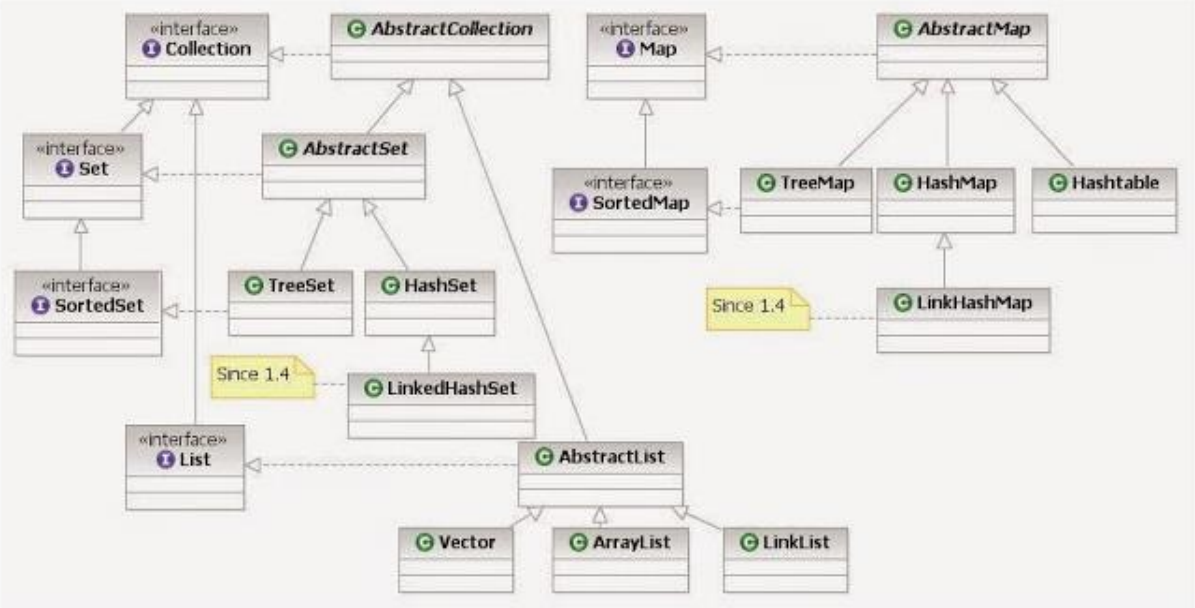
Q3 What is the difference between Collection and Collections ?

Collection is an interface while Collections is a java class , both are present in java.util package and part of java collections framework.

Q4 Which collection classes are synchronized or thread-safe ?

Stack, Properties , Vector and Hashtable can be used in multi threaded environment because they are synchronized classes (or thread-safe).

Q5 Name the core Collection interfaces ?



Q9 What is an iterator ?

Iterator is an interface . It is found in java.util package. It provides methods to iterate over any Collection.

Q10 What is the difference between Iterator and Enumeration ?

The main difference between Iterator and Enumeration is that Iterator has remove() method while Enumeration doesn't.
Hence , using Iterator we can manipulate objects by adding and removing the objects from the collections. Enumeration behaves like a read only interface as it can only traverse the objects and fetch it .

Q14 How to reverse the List in Collections ?

There is a built in reverse method in Collections class . reverse(List list) accepts list as parameter.

Collections.reverse(listobject);

Q15 How to convert the array of strings into the list ?

Arrays class of java.util package contains the method asList() which accepts the array as parameter.
So,

```
String[]  wordArray =  {"Love Yourself"  , "Alive is Awesome" , "Be in present"};
List wordList =  Arrays.asList(wordArray);
```

Q19 What is the difference between Iterator and ListIterator.

Using Iterator we can traverse the list of objects in forward direction . But ListIterator can traverse the collection in both directions that is forward as well as backward.

Q21 What is the difference between HashSet and TreeSet ?

Main differences between HashSet and TreeSet are :
a. HashSet maintains the inserted elements in random order while TreeSet maintains elements in the sorted order
b. HashSet can store null object while TreeSet can not store null object.

Q28 Why Map interface does not extend the Collection interface in Java Collections Framework ?

One liner answer : **Map interface is not compatible with the Collection interface.**
Explanation : Since Map requires key as well as value , for example , if we want to add key-value pair then we will use put(Object key , Object value) . So there are two parameters required to add element to the HashMap object . In Collection interface add(Object o) has only one parameter.
The other reasons are Map supports valueSet , keySet as well as other appropriate methods which have just different views from the Collection interface.

Q32 What is CopyOnWriteArrayList ? How it is different from ArrayList in Java?

CopyOnWriteArrayList is a thread safe variant of ArrayList in which all mutative operations like add , set are implemented by creating a fresh copy of the underlying array.
It guaranteed not to throw ConcurrentModificationException.
It permits all elements including null. It is introduced in jdk 1.5 .

Q35 What is BlockingQueue in Java Collections Framework?

BlockingQueue implements the java.util.Queue interface . BlockingQueue supports operations that wait for the queue to become non-empty when retrieving an element , and wait for space to become available in the queue when storing an element .
It does not accept null elements.
Blocking queues are primarily designed for the producer-consumer problems.
BlockingQueue implementations are thread-safe and can also be used in inter-thread communications.
This concurrent Collection class was added in jdk 1.5

Q38 What is the difference between Fail- fast iterator and Fail-safe iterator ?

This is one of the most popular interview question for the higher experienced java developers .
Main differences between Fail-fast and Fail-safe iterators are :
a. Fail-fast throw ConcurrentModificationException while Fail-safe does not.
b. Fail-fast does not clone the original collection list of objects while Fail-safe creates a copy of the original collection list of objects.
The difference is explained in detail here fail-safe vs fail-fast iterator in java.

Q48 How will you make Collections readOnly ?

We can make the Collection readOnly by using the following lines code:

General : Collections.unmodifiableCollection(Collection c)

Collections.unmodifiableMap(Map m)
Collections.unmodifiableList(List l)
Collections.unmodifiableSet(Set s)

Q49 What is UnsupportedOperationException?

This exception is thrown to indicate that the requested operation is not supported.
Example of UnsupportedOperationException:
In other words, if you call add() or remove() method on the readOnly collection . We know readOnly collection can not be modified . Hence ,
UnsupportedOperationException will be thrown.

Q3 Why HashSet does not have get(Object o) method ?

Most of the people get puzzled by hearing this question . This question tests the deep understanding of the HashSet class .This question helps the interviewer to know whether candidate has the idea about contains() method in HashSet class or not .So let jump to the answer

get(Object o) is useful when we have one information linked to other information just like key value pair found in HashMap .So using get() method on one information we can get the second information or vice-versa.

Unlike HashMap , HashSet is all about having unique values or unique objects . There is no concept of keys in HashSet .
The only information we can derive from the HashSet object is whether the element is present in the HashSet Object or not . If the element is not present in the HashSet then add it otherwise return true leaving HashSet object unchanged. Here, contains() method helps to provide this information.

Due to the above reason there is no get(Object o) method in HashSet.

Q6 What is the default initial capacity and initial load factor of HashSet object?

As we already discussed that HashSet internally uses HashMap . So the default initial capacity and initial load factor of HashSet is same as that of HashMap , that is

Default Initial Capacity of HashSet Object : 16
Initial Load Factor of HashSet Object : 0.75

Iteration performance of the HashSet object depends on the above two factors that is initial capacity and load factor :

a. It is very important not to set the initial capacity too high or the load factor too low if iteration performance is important.

3. What is the difference between fail-fast and fail-safe Iterators? (answer)

This is relatively *new collection interview questions* and can become trick if you hear the term fail-fast and fail-safe first time. Fail-fast Iterators throws `ConcurrentModificationException` when one Thread is iterating over collection object and other thread structurally modify Collection either by adding, removing or modifying objects on underlying collection. They are called fail-fast because they try to immediately throw Exception when they encounter failure. On the other hand fail-safe Iterators works on copy of collection instead of original collection

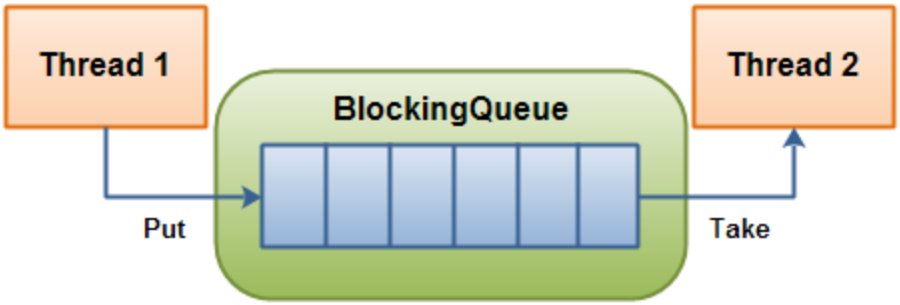
4. How do you remove an entry from a Collection? and subsequently what is the difference between the `remove()` method of `Collection` and `remove()` method of `Iterator`, which one you will use while removing elements during iteration?

Collection interface defines `remove(Object obj)` method to remove objects from Collection. List interface adds another method `remove(int index)`, which is used to remove object at specific index. You can use any of these method to remove an entry from Collection, while not iterating. Things change, when you iterate. Suppose you are traversing a List and removing only certain elements based on logic, then you need to use Iterator's `remove()` method. This method removes current element from Iterator's perspective. If you use Collection's or List's `remove()` method during iteration then your code will throw `ConcurrentModificationException`. That's why it's advised to use Iterator remove() method to remove objects from Collection.

BlockingQueue: A blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying 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.

The java.util.concurrent package contains a set of synchronized Queue interfaces and classes. Blocking Queue extends Queue with operations that wait for the queue to become nonempty when retrieving an element and for space to become available in the queue when storing an element.

Here is a diagram showing two threads cooperating via a blocking queue:



A BlockingQueue with one thread putting into it, and another thread taking from it.

Following list of points about BlockingQueue in Java will help to learn and understand more about it:

1) BlockingQueue in Java doesn't allow null elements, various implementation
of BlockingQueue like ArrayBlockingQueue, LinkedBlockingQue ue throws NullPointerException when you try to add null on

queue.

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<String>(10);
//bQueue.put(null); //NullPointerException - BlockingQueue in Java doesn't allow null

bQueue = new LinkedBlockingQueue<String>();
bQueue.put(null);

Exception in thread "main" java.lang.NullPointerException
        at java.util.concurrent.LinkedBlockingQueue.put(LinkedBlockingQueue.java:288)
```

2) `BlockingQueue` can be bounded or unbounded. A bounded `BlockingQueue` is one which is initialized with initial capacity and call to `put()` will be blocked if `BlockingQueue` is full and size is equal to capacity. This bounding nature makes it ideal to use a shared queue between multiple threads like in most common Producer consumer solutions in Java. An unbounded Queue is one which is initialized without capacity, actually by default it initialized with `Integer.MAX_VALUE`. most common example of `BlockingQueue` uses **bounded** `BlockingQueue` as shown in below example.

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<String>(2);
bQueue.put("Java");
System.out.println("Item 1 inserted into BlockingQueue");
bQueue.put("JDK");
System.out.println("Item 2 is inserted on BlockingQueue");
bQueue.put("J2SE");
System.out.println("Done");

Output:
Item 1 inserted into BlockingQueue
Item 2 is inserted on BlockingQueue
```

This code will only insert Java and JDK into `BlockingQueue` and then it will block while inserting 3rd element J2SE because size of `BlockingQueue` is 2 here.

3)`BlockingQueue` implementations like `ArrayBlockingQueue`, `LinkedBlockingQueue` and `PriorityBlockingQueue` are thread-safe. All queuing method uses concurrency control and internal locks to perform operation atomically. Since `BlockingQueue` also extend Collection, bulk Collection operations like `addAll()`, `containsAll()` are not performed atomically until any `BlockingQueue` implementation specifically supports it. So call to `addAll()` may fail after inserting couple of elements.

4) Common methods of `BlockingQueue` is are `put()` and `take()` which are blocking methods in Java and used to insert and retrive elements from `BlockingQueue` in Java. `put()` will block if `BlockingQueue` is full and `take()` will block if `BlockingQueue` is empty, call to `take()` removes element from head of Queue as shown in following example:

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<String>(2);
bQueue.put("Java"); //insert object into BlockingQueue
System.out.println("BlockingQueue after put: " + bQueue);
bQueue.take(); //retrieve object from BlockingQueue in Java
System.out.println("BlockingQueue after take: " + bQueue);
```

```
Output:
BlockingQueue after put: [Java]
BlockingQueue after take: []
```

5) `BlockingQueue` interface extends `Collection`, `Queue` and `Iterable` interface which provides it all `Collection` and `Queue` related methods like `poll()`, and `peak()`, unlike `take()`, `peek()` method returns head of the queue without removing it, `poll()` also retrieves and removes elements from head but can wait till specified time if `Queue` is empty.

```
BlockingQueue<String> linkedBQueue = new LinkedBlockingQueue<String>(2);
linkedBQueue.put("Java"); //puts object into BlockingQueue
System.out.println("size of BlockingQueue before peek : " + linkedBQueue.size());
System.out.println("example of peek() in BlockingQueue: " + linkedBQueue.peek());
System.out.println("size of BlockingQueue after peek : " + linkedBQueue.size());
System.out.println("calling poll() on BlockingQueue: " + linkedBQueue.poll());
System.out.println("size of BlockingQueue after poll : " + linkedBQueue.size());
```

```
Output:
size of BlockingQueue before peek : 1
example of peek() in BlockingQueue: Java
size of BlockingQueue after peek : 1
calling poll() on BlockingQueue: Java
size of BlockingQueue after poll : 0
```

6)Other important methods from `BlockingQueue` in Java is `remainingCapacity()` and `offer()`, former returns number remaining space in `BlockingQueue`, which can be filled without blocking while later insert object into queue if possible and return true if success and false if fail unlike `add()` method which throws `IllegalStateException` if it fails to insert object into `BlockingQueue`. Use `offer()` over `add()` wherever possible.

Usage of `BlockingQueue` in Java

There can be many creative usage of `BlockingQueue` in Java given its flow control ability. Two of the most common ways I see programmer uses `BlockingQueue` is to implement Producer Consumer design pattern and implementing Bounded buffer in Java. It surprisingly made coding and inter thread communication over a shared object very easy.

`ArrayBlockingQueue` and `LinkedBlockingQueue` in Java

`ArrayBlockingQueue` and `LinkedBlockingQueue` are common implementation of `BlockingQueue<E>` interface. `ArrayBlockingQueue` is backed by array and `Queue` impose orders as FIFO. head of the queue is the oldest element in terms of time and tail of the queue is youngest element. `ArrayBlockingQueue` is also fixed size bounded buffer on the other hand `LinkedBlockingQueue` is an optionally bounded queue built on top of Linked nodes. In terms of throughput `LinkedBlockingQueue` provides higher throughput than Array`BlockingQueue` in Java.

Methods in Queue:

1. **add()-** This method is used to add elements at the tail of queue. More specifically, at the last of linkedlist if it is used, or according to the priority in case of priority queue implementation.
2. **peek()-** This method is used to view the head of queue without removing it. It returns Null if the queue is empty.
3. **element()-** This method is similar to peek(). It throws *NoSuchElementException* when the queue is empty.
4. **remove()-** This method removes and returns the head of the queue. It throws *NoSuchElementException* when the queue is impty.
5. **poll()-** This method removes and returns the head of the queue. It returns null if the queue is empty.

This interface is implemented by the following classes:

- **LinkedBlockingQueue** — an optionally bounded FIFO blocking queue backed by linked nodes
- **ArrayBlockingQueue** — a bounded FIFO blocking queue backed by an array
- **PriorityBlockingQueue** — an unbounded blocking priority queue backed by a heap
- **DelayQueue** — a time-based scheduling queue backed by a heap
- **SynchronousQueue** — a simple rendezvous mechanism that uses the BlockingQueue interface

Real World Example of Producer Consumer Design Pattern

Producer consumer pattern is every where in real life and depict coordination and collaboration. Like one person is preparing food (Producer) while other one is serving food (Consumer), both will use shared table for putting food plates and taking food plates. Producer which is the person preparing food will wait if table is full and Consumer (Person who is serving food) will wait if table is empty. table is a shared object here. On Java library **Executor framework** itself implement Producer Consumer design pattern be separating responsibility of addition and execution of task.

Problem

To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Benefit of Producer Consumer Pattern

Its indeed a useful design pattern and used most commonly while writing multi-threaded or concurrent code. here is few of its benefit:

1) Producer Consumer Pattern simple development. you can Code Producer and Consumer independently and Concurrently, they just need to know shared object.

2) Producer doesn't need to know about who is consumer or how many consumers are there. Same is true with Consumer.

3) Producer and Consumer can work with different speed. There is no risk of Consumer consuming half-baked item. In fact by monitoring consumer speed one can introduce more consumer for better utilization.

4) Separating producer and Consumer functionality result in more clean, readable and manageable code.

Producer Consumer Problem in Multi-threading

Producer-Consumer Problem is also a popular java interview question where interviewer ask to implement producer consumer design pattern so that Producer should wait if Queue or bucket is full and Consumer should wait if queue or bucket is empty. This problem can be implemented or solved by different ways in Java, classical way is using wait and notify method to communicate between **Producer and Consumer thread** and blocking each of them on individual condition like full queue and empty queue. With introduction of **BlockingQueue** Data Structure in Java 5 Its now much simpler because BlockingQueue provides this control implicitly by introducing blocking methods put() and take(). Now you don't require to use wait and notify to communicate between Producer and Consumer. BlockingQueue put() method will block if Queue is full in case of Bounded Queue and take() will block if Queue is empty. In next section we will see a *code example of Producer Consumer design pattern*.

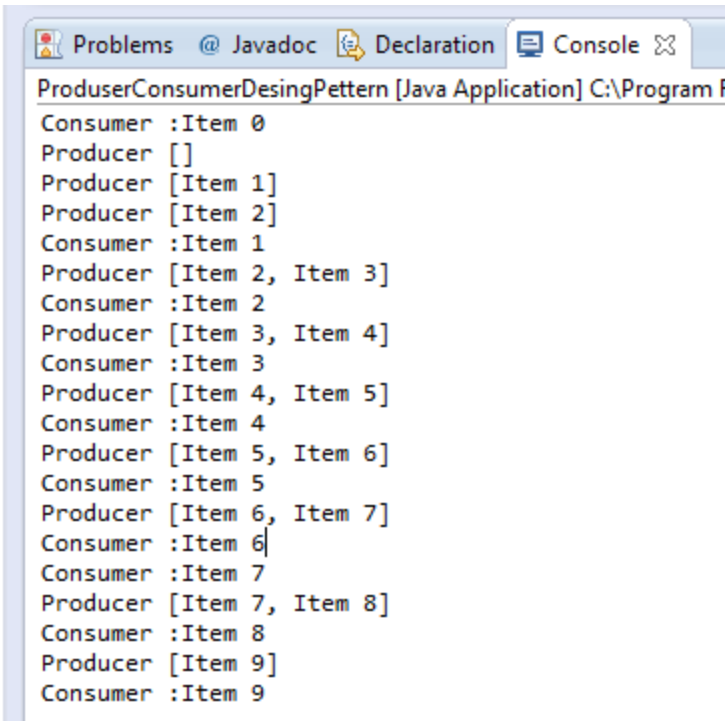
Using Blocking Queue to implement Producer Consumer Pattern

BlockingQueue amazingly simplifies implementation of Producer-Consumer design pattern by providing outofbox support of blocking on put() and take(). Developer doesn't need to write confusing and critical piece of wait-notify code to implement communication. `BlockingQuue` is an interface and Java 5 provides different implantation like `ArrayBlockingQueue` and `LinkedBlockingQueue` , both implement FIFO order or elements, while `ArrayLinkedQueue` is bounded in nature `LinkedBlockingQueue` is optionally bounded. here is a complete **code example of Producer Consumer pattern** with BlockingQueue. Compare it with classic wait notify code, its much simpler and easy to understand.

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class ProduserConsumerDesingPettern {

    public static void main(String[] args) {

        BlockingQueue<String> blockingQueue = new LinkedBlockingQueue<String>();

        Thread produ = new Thread(new ProducerThread(blockingQueue));
        Thread consu = new Thread(new ConsumerThread(blockingQueue));

        produ.start();
        consu.start();
    }

}
```

```java
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class ProducerThread implements Runnable {

    BlockingQueue<String> blockingQueue = new LinkedBlockingQueue<>();

    public ProducerThread(BlockingQueue<String> blockingQueue) {

        this.blockingQueue = blockingQueue;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                blockingQueue.put("Item " + i);
                System.out.println("Producer " + blockingQueue);
            } catch (InterruptedException e) {

                e.printStackTrace();
            }

        }
    }

}
```

```java
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class ConsumerThread implements Runnable {
    BlockingQueue<String> blockingQueue = new LinkedBlockingQueue<>();

    public ConsumerThread(BlockingQueue<String> blockingQueue) {

        this.blockingQueue = blockingQueue;
    }

    @Override
    public void run() {
        while (true) {

            try {
                System.out.println("Consumer :" + blockingQueue.take());
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

        }

    }
}
```



You see Producer Thread produced number and Consumer thread consumes it in FIFO order because blocking queue allows elements to be accessed in FIFO.

Program to Solve the Producer-Consumer Problem using wait and notify

```
package com.pr;
  import java.util.ArrayList;
  import java.util.List;
  public class ProducerConsumer {
       public static void main (String[] args) {
            List<Integer> sharedQueue = new ArrayList<Integer>();
            Producer producer = new Producer(sharedQueue);
            Consumer consumer = new Consumer(sharedQueue);
            Thread p = new Thread(producer, "Producer Thread");
            Thread c = new Thread(consumer, "Consumer Thread");
            p.start();
            c.start();
       }
  }

  class Producer implements Runnable {
       List<Integer> sharedQueue = new ArrayList<Integer>();
       public Producer(List<Integer> sharedQueue) {
            this.sharedQueue = sharedQueue;
       }
       public void run() {
            for (int i = 1; i<=10; i++) {
                 try {
                      produce(i);
                 } catch(InterruptedException e) {
                      e.printStackTrace();
                 }
            }
       }
```

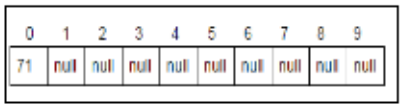
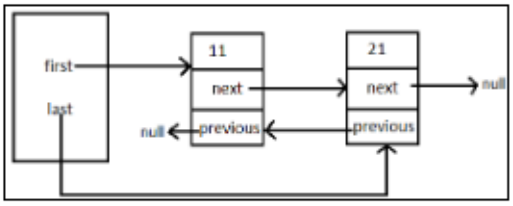
```
    private void produce(int i) throws InterruptedException{
        synchronized (sharedQueue) {
            if (sharedQueue.size() == 1) {
                System.out.println("Queue is full");
                sharedQueue.wait();
            }
        }
        synchronized (sharedQueue) {
            System.out.println("Produced : "+i);
            sharedQueue.add(i);
            Thread.sleep(1000);
            sharedQueue.notify();
        }
    }
}

class Consumer implements Runnable {
    List<Integer> sharedQueue = new ArrayList<Integer>();
    public Consumer(List<Integer> sharedQueue) {
        this.sharedQueue = sharedQueue;
    }
    @Override
    public void run() {
        while (true) {
            try {
                consume();
                Thread.sleep(1000);
            } catch(InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

```
        private void consume() throws InterruptedException{
            synchronized (sharedQueue) {
                while (sharedQueue.size() == 0) {
                    System.out.println("Queue is empty");
                    sharedQueue.wait();
                }
            }
            synchronized (sharedQueue) {
                Thread.sleep(1000);
                System.out.println("Consumed :" +sharedQueue.remove(0));
                sharedQueue.notify();
            }
        }
    }
```

Output : - Produced : 1
Queue is full
Consumed : 1
Produced : 2
Queue is full
Consumed : 2
Produced : 3
Queue is full
Consumed : 3
Produced : 4
Queue is full
Consumed : 4
Produced : 5
Consumed : 5
Produced : 6
Queue is full
Consumed : 6
Produced : 7
Consumed : 7
Produced : 8
Consumed : 8
Produced : 9

Queue is full
Consumed : 9
Produced : 10
Consumed : 10
Queue is empty

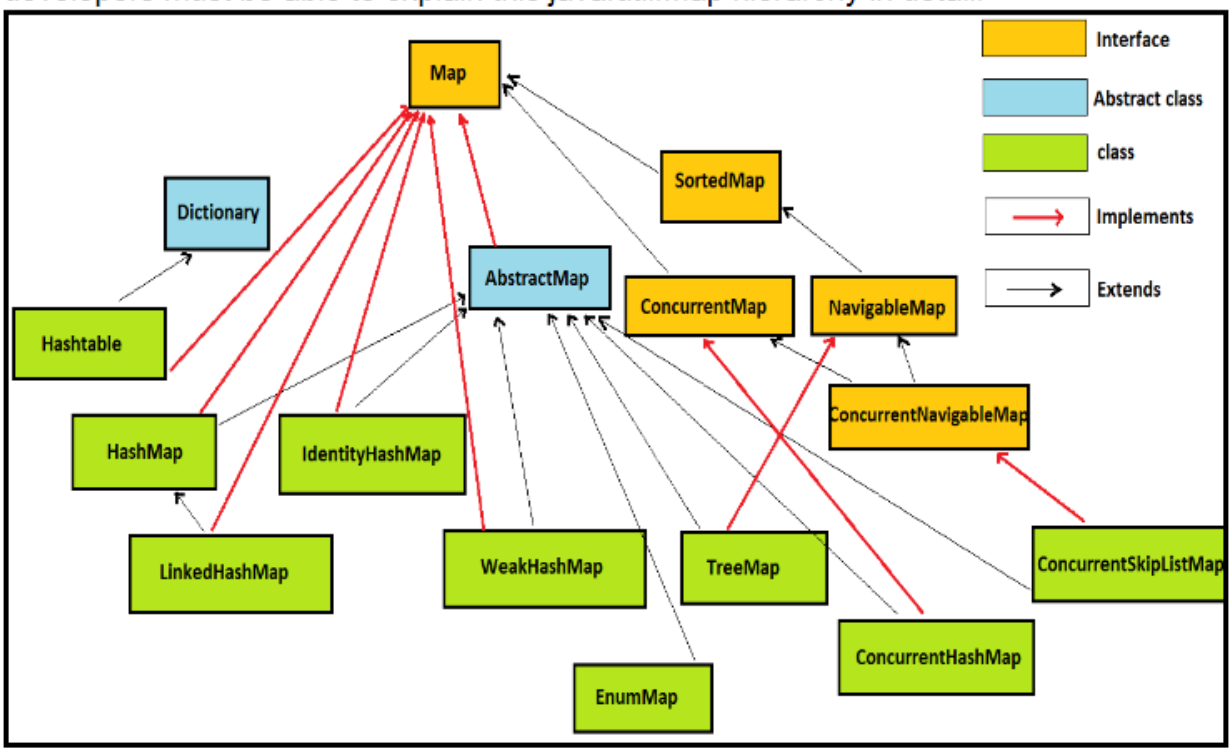
What are differences between ArrayList and LinkedList in java?

	Property	***java.util.ArrayList***	**java.util.LinkedList**
1	Structure	java.util.ArrayList is index based structure in java. 	A java.util.**LinkedList** is a data structure consisting of a group of **nodes** which together represent a sequence. node is composed of a data and a reference (in other words, a **link**) to the next node in the sequence in java. 
2	**Resizable**	**ArrayList is Resizable-array in java.**	New node is created for storing new element in LinkedList in java.
3	**Initial capacity**	java.util.ArrayList is created with initial capacity of 10 in java.	For storing every element node is created in LinkedList, so linkedList's initial capacity is 0 in java.
4	Ensuring **Capacity**/ resizing.	ArrayList is created with initial capacity of 10. ArrayList's size is **increased by 50%** i.e. after resizing it's size become 15 in java.	For storing every element node is created, so linkedList's initial capacity is 0, it's size grow with addition of each and every element in java.

5	RandomAccess interface	ArrayList implements RandomAccess(Marker interface) to indicate that they support fast random access (i.e. index based access) in java.	LinkedList does not implement RandomAccess interface in java.
6	AbstractList and AbstractSequentialList	ArrayList extends AbstractList (abstract class) which provides implementation to List interface to minimize the effort required to implement this interface backed by RandomAccess interface.	LinkedList extends AbstractSequentialList (abstract class), AbstractSequentialList extends AbstractList. In LinkedList, data is accessed sequentially, so for obtaining data at specific index, iteration is done on nodes sequentially in java.
7	How **get(index)** method works? (Though difference has been discussed briefly in above 2 points but in this in point we will figure difference in detail.)	Get method of ArrayList directly gets element on specified index. Hence, offering O(1) complexity in java.	Get method of LinkedList iterates on nodes sequentially to get element on specified index. Hence, offering O(n) complexity in java.
8	**When to use**	**Use ArrayList when get operations is more frequent than add and remove operations in java.**	**Use LinkedList when add and remove operations are more frequent than get operations in java.**

Collection interview Question 11. What are core classes and interfaces in java.util.Map hierarchy?

Answer. Freshers must know core classes in Map hierarchy but experienced developers must be able to explain this java.util.Map hierarchy in detail.



***Differences** between java.util.**Iterator** and java.util.**Enumeration** in java >*

	Property	***java.util.Enumeration***	***java.util.Iterator***
1	Remove elements during iteration	java.util.Enumeration **doesn't allows** to remove elements from collection during iteration in java.	java.util.Iterator **allows** to remove elements from collection during iteration by using **remove()** method in java.
2	Improved naming conventions in Iterator	**nextElement()** Method Returns the next element of this enumeration if this enumeration object has at least one more element to provide. **hasMoreElements()** returns true if enumeration contains more elements.	**nextElement()** has been changed to **next()** in Iterator. And **hasMoreElements()** has been changed to **hasNext()** in Iterator.
3	Introduced in which java version	Enumeration was introduced in first version of java i.e. **JDK 1.0**	Iterator was introduced in second version of java i.e. **JDK 2.0** Iterator was introduced to replace Enumeration in the Java Collections Framework.
4	Recommendation	**Java docs** recommends iterator over enumeration**.**	**Java docs** recommends iterator over enumeration**.**
5	Enumeration and Iterator over **Vector**	**Enumeration** returned by Vector is **fail-safe**, means any modification made to Vector during iteration using Enumeration don't throw any exception in java.	**Iterator** returned by Vector are **fail-fast**, means any structural modification made to ArrayList during iteration will throw ConcurrentModificationException in java.

Differences between java.util.HashMap and java.util.Hashtable in java >

	Property	***java.util.HashMap***	***java.util.Hashtable***
1	synchronization	java.util.HashMap is **not synchronized** (because 2 threads on same HashMap object can access it at same time) in java.	java.util.Hashtable is **synchronized** (because 2 threads on same Hashtable object cannot access it at same time) in java.
2	Performance	HashMap is not synchronized, hence its operations are **faster** as compared to Hashtable in java.	Hashtable is synchronized, hence its operations are **slower** as compared to HashMap in java. If we are working not working in multithreading environment jdk recommends us to use HashMap.
3	Null keys and values	HashMap allows to store **one null key** and **many null values** i.e. many keys can have null value in java.	Hashtable does **not allow to store null key or null value**. Any attempt to store null key or value throws runtimeException (NullPointerException) in java.
4	Introduced in which java version	HashMap was introduced in second version of java i.e. **JDK 2.0**	Hashtable was introduced in first version of java i.e. **JDK 1.0** But it was refactored in java 2 i.e. JDK 1.2 to implement the Map interface, hence making it a member of member of the Java Collections Framework.
5	Recommendation	In non-multithreading environment it is recommended to use HashMap than using Hashtable in java.	In **java 5 i.e. JDK 1.5**, it is **recommended** to use ConcurrentHashMap than using Hashtable.
6	Extends Dictionary (Abstract class, which is obsolete)	HashMap does not extends Dictionary in java.	Hashtable extends Dictionary (which maps non-null keys to values. In a given Dictionary we can look up value corresponding to key) in java.

***Differences** between java.util.**HashMap** and java.util.concurrent.ConcurrentHashMap in java >*

Property	*java.util.**HashMap***	*java.util.concurrent.* ***ConcurrentHashMap***
synchronization	HashMap is **not synchronized.**	ConcurrentHashMap is **synchronized**.
2 threads on same Map object can access it at concurrently?	Yes, because HashMap is not synchronized.	Yes. But how despite of being synchronized, 2 threads on same *ConcurrentHashMap* object can access it at same time? *ConcurrentHashMap* is divided into different **segments** based on concurrency level. So different threads can access different **segments** concurrently.

Performance	We will **synchronize HashMap and then compare its performance with ConcurrentHashMap**. *We can synchronize hashMap by using Collections's class* ***synchronizedMap*** *method.* `Map synchronizedMap = Collections.synchronizedMap(hashMap);` *Now, no 2 threads can access same instance of map concurrently.* **Hence synchronized HashMap's performance is slower as compared to ConcurrentHashMap.** But why we didn't compared HashMap (unSynchronized) with ConcurrentHashMap? Because performance of unSynchronized collection is always better than some synchronized collection. As, default (unSynchronized) hashMap didn't cause any locking.	**ConcurrentHashMap's performance is faster as compared to HashMap (**because it is divided into segments, as discussed in above point**).** *Read this post for performance comparison between HashMap and ConcurrentHashMap.*

Null keys and values	HashMap allows to store **one null key** and **many null values** i.e. any key can have null value.	ConcurrentHashMap does **not allow to store null key or null value**. Any attempt to store null key or value throws runtimeException (NullPointerException).
iterators	The iterators returned by the iterator() method of HashMap are ***fail-fast*** > *hashMap.keySet().iterator()* *hashMap.values().iterator()* *hashMap.entrySet().iterator()* all three iterators are ***fail-fast***	iterators are ***fail-safe***. *concurrentHashMap.keySet().iterator()* *concurrentHashMap.values().iterator()* *concurrentHashMap.entrySet().iterator()* all three iterators are ***fail-safe.***

putIfAbsent	HashMap does not contain putIfAbsent method.
	putIfAbsent *method is equivalent to writing following code >*
	
	`synchronized (map){`
	`  if (!map.containsKey(key))`
	`  return map.put(key, value);`
	`  else`
	`  return map.get(key);`
	`}`
	
	Program to create method that provides functionality similar to putIfAbsent method of ConcurrentHashMap and to be used with HashMap
		If map does not contain specified `key`, put specified `key-value` pair in map and return null.
		If map already contains specified `key`, return value corresponding to specified `key`.
		
		Program to use ConcurrentHashMap's putIfAbsent method

Introduced in which java version	HashMap was introduced in **java 2 i.e. JDK 1.2**,	ConcurrentHashMap was introduced in **java 5** i.e. **JDK 1.5**, since its introduction Hashtable has become obsolete, because of concurrency level its performance is better than Hashtable.
Implements which interface	HashMap implements **java.util.Map**	ConcurrentHashMap implements **java.util.Map** and **java.util.concurrent.ConcurrentMap**
Package	HashMap is in **java.util** package	ConcurrentHashMap is in **java.util.concurrent** package.

List	Duplicate elements	insertion order	Sorted by natural order	synchronized	null elements	Iterator
ArrayList	Yes	Yes			Yes	Iterator & listIterator are Fail-fast
LinkedList	Yes	Yes			Yes	Iterator & listIterator are Fail-fast
CopyOnWriteArrayList	Yes	Yes		Yes	Yes	Iterator & listIterator are **Fail-safe**

Set	Duplicate elements	insertion order	Sorted by natural order	synchronized	null elements	Iterator
HashSet					Yes	Fail-fast
LinkedHashSet		Yes			Yes	Fail-fast
TreeSet			Yes		No	Fail-fast
ConcurrentSkipListSet			Yes	Yes	No	**Fail-safe**

Map	Duplicate Keys	insertion order of keys	Sorted by natural order of keys	synchronized	null keys or null values	Iterator Map implementations returns 3 iterators > *map.keySet().iterator()* *map.values().iterator()* *map.entrySet().iterator()*
HashMap					one null key and many null values	All are Fail-fast
Hashtable				Yes	No	All are Fail-fast
ConcurrentHashMap				Yes	No	All are **Fail-safe**
TreeMap			Yes		Null key not allowed, Allow many null values	All are Fail-fast
ConcurrentSkipListMap			Yes	Yes	No	All are **Fail-safe**

Question 6. What Is A Diamond Operator?

Answer : Diamond operator let the compiler infer the type arguments for the generic classes. It is added in Java 7.

As Example - Before JDK7 if we had to define a Map using String as both Key and Value we had to write it like this -

Map<String, String> cityMap = new HashMap<String, String>();

In Java SE 7, you can substitute the parameterized type of the constructor with an empty set of type parameters (<>) known as diamond operator.

Map<String, String> cityMap = new HashMap<>();

Question 9. What Is Randomaccess Interface?

Answer : RandomAccess interface is a marker interface used by List implementations to indicate that they support fast (generally constant time) random access.

Question 17. How Arraylist Works Internally In Java?

Answer : Basic data structure used by ArrayList to store objects is an array of Object class, which is defined like -

transient Object[] elementData;

When we add an element to an ArrayList it first verifies whether it has that much capacity in the array to store new element or not, in case there is not then the new capacity is calculated which is 50% more than the old capacity and the array is increased by that much capacity (Actually uses Arrays.copyOf which returns the original array increased to the new length)

8. What is the difference between Iterator and ListIterator?

Iterator	ListIterator
Iterator traverses the elements in forward direction only.	ListIterator traverses the elements in both directions(forward & backward).
Iterator can be used with List, Set and Queue.	ListIterator can be used with List only.

Additionally, ListIterator comes with extra functionalities like adding an element, replacing an element, getting index position for previous and next elements.