1. What are the main differences between array and collection?

Arrays	Collection
Arrays are fixed in size that is once we create an array we cannot increased or decreased based on our requirement.	Collection are growable in nature that is based on our requirement. We can increase or decrease of size.
Not recommended in terms of memory	Recommended in terms of memory
Better in terms of performance	Lagging in terms of performance
Can hold only homogeneous datatypes	Can hold both homogenous and heterogeneous data.
There is no underlying data structure for arrays and hence readymade method support is not available.	Every collection class is implemented based on some standard data structure and hence for every requirement readymade method support is available being a performance. We can use these methods directly and we are not responsible to implement these methods.
Arrays can hold both object and primitive.	Collection can hold only object types but primitive.

2. Explain various interfaces used in Collection framework?

The Collections framework has a lot of Interfaces, setting the fundamental nature of various collection classes.

The Collection Interface

It is at the top of collection hierarchy and must be implemented by any class that defines a collection. Its general declaration is,

```
interface Collection <E>
```

Collection Interface Methods

1. Following are some of the commonly used methods in this interface.

Methods	Description
boolean add(E obj)	Used to add objects to a collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates.
boolean addAll(Collection C)	Add all elements of collection C to the invoking collection. Returns true if the element were added. Otherwise, returns false.
boolean remove(Object obj)	To remove an object from collection. Returns true if the element was removed. Otherwise, returns false.
boolean removeAll(Collection C)	Removes all element of collection C from the invoking collection. Returns true if the collection's elements were removed. Otherwise, returns false.
boolean contains(Object obj)	To determine whether an object is present in collection or not. Returns true if obj is an element of the invoking collection. Otherwise, returns false.

boolean isEmpty()	Returns true if collection is empty, else returns false.
int size()	Returns number of elements present in collection.
void clear()	Removes total number of elements from the collection.
Object[] toArray()	Returns an array which consists of the invoking collection elements.
boolean retainAll(Collection c)	Deletes all the elements of invoking collection except the specified collection.
Iterator iterator( )	Returns an iterator for the invoking collection.
boolean equals(Object obj)	Returns true if the invoking collection and obj are equal. Otherwise, returns false.
Object[] toArray(Object array[])	Returns an array containing only those collection elements whose type matches of the specified array.

The List Interface

It extends the **Collection** Interface, and defines storage as sequence of elements. Following is its general declaration,

```
interface List <E>
```

1. Allows random access and insertion, based on position.
2. It allows Duplicate elements.

List Interface Methods

3. Apart from methods of Collection Interface, it adds following methods of its own.

Methods	Description
Object get(int index)	Returns object stored at the specified index
Object set(int index, E obj)	Stores object at the specified index in the calling collection
int indexOf(Object obj)	Returns index of first occurrence of obj in the collection
int lastIndexOf(Object obj)	Returns index of last occurrence of obj in the collection
List subList(int start, int end)	Returns a list containing elements between start and end index in the collection

The Set Interface

This interface defines a Set. It extends **Collection** interface and doesn't allow insertion of duplicate elements. It's general declaration is,

```
interface Set <E>
```

1. It doesn't define any method of its own. It has two sub interfaces, **SortedSet** and **NavigableSet**.
2. **SortedSet** interface extends **Set** interface and arranges added elements in an ascending order.

3. **NavigabeSet** interface extends **SortedSet** interface, and allows retrieval of elements based on the closest match to a given value or values.

The Queue Interface

It extends **collection** interface and defines behaviour of queue, that is first-in, first-out. It's general declaration is,

```
interface Queue <E>
```

Queue Interface Methods

There are couple of new and interesting methods added by this interface. Some of them are mentioned in below table.

Methods	Description
Object poll()	removes element at the head of the queue and returns **null** if queue is empty
Object remove()	removes element at the head of the queue and throws **NoSuchElementException** if queue is empty
Object peek()	returns the element at the head of the queue without removing it. Returns **null** if queue is empty
Object element()	same as peek(), but throws **NoSuchElementException** if queue is empty
boolean offer(E obj)	Adds object to queue.

The Dequeue Interface

It extends **Queue** interface and implements behaviour of a double-ended queue. Its general declaration is,

```
interface Dequeue <E>
```

1. Since it implements Queue interface, it has the same methods as mentioned there.
2. Double ended queues can function as simple queues as well as like standard Stacks.

3. What is the difference 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.

4. What is the difference between ArrayList and LinkedList?

ArrayList	LinkedList
This class uses a dynamic array to store the elements in it. With the introduction of generics, this class supports the storage of all types of objects.	This class uses a doubly linked list to store the elements in it. Similar to the ArrayList, this class also supports the storage of all types of objects.
Manipulating ArrayList takes more time due to the internal implementation. Whenever we remove an element, internally, the array is traversed and the memory bits are shifted.	Manipulating LinkedList takes less time compared to ArrayList because, in a doubly-linked list, there is no concept of shifting the memory bits. The list is traversed and the reference link is changed.
This class implements a List interface. Therefore, this acts as a list.	This class implements both the List interface and the Deque interface. Therefore, it can act as a list and a deque.
This class works better when the application demands storing the data and accessing it.	This class works better when the application demands manipulation of the stored data.

5. What is the difference between Iterator and ListIterator?

Iterator	ListIterator
Can traverse elements present in Collection only in the forward direction.	Can traverse elements present in Collection both in forward and backward directions.
Helps to traverse Map, List and Set.	Can only traverse List and not the other two.
Indexes cannot be obtained by using Iterator.	It has methods like nextIndex() and previousIndex() to obtain indexes of elements at any time while traversing List.
Cannot modify or replace elements present in Collection	We can modify or replace elements with the help of set(E e)
Cannot add elements and it throws ConcurrentModificationException.	Can easily add elements to a collection at any time.
Certain methods of Iterator are next(), remove() and hasNext().	Certain methods of ListIterator are next(), previous(), hasNext(), hasPrevious(), add(E e).

6. What is the difference between List and a Set?

List	Set
The List is an ordered sequence.	The Set is an unordered sequence.
List allows duplicate elements	Set doesn’t allow duplicate elements.
Elements by their position can be accessed.	Position access to elements is not allowed.
Multiple null elements can be stored.	Null element can store only once.
List implementations are ArrayList, LinkedList, Vector, Stack	Set implementations are HashSet, LinkedHashSet.

7. What are the differences between HashSet and TreeSet?

Hash Set	Tree Set
Hash set is implemented using HashTable	The tree set is implemented using a tree structure.
HashSet allows a null object	The tree set does not allow the null object. It throws the null pointer exception.
Hash set use equals method to compare two objects	Tree set use compare method for comparing two objects.
Hash set doesn't now allow a heterogeneous object	Tree set allows a heterogeneous object
HashSet does not maintain any order	TreeSet maintains an object in sorted order

8. What is the difference between HashSet and HashMap?

Basic	HashSet	HashMap
Implements	Set interface	Map interface
Duplicates	No	Duplicates values are allowed but no duplicate key is allowed
Dummy values	Yes	No
Objects required during an add operation	1	2
Adding and storing mechanism	HashMap object	Hashing technique
Speed	It is comparatively slower than HashMap	It is comparatively faster than HashSet because of hashing technique has been used here.
Null	Have a single null value	Single null key and any number of null values
Insertion Method	add()	put()

9. What is the difference between HashMap and HashTable?

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.

10. What is the difference between Comparable and Comparator?

Comparable	Comparator
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.
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.
Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
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.

11. How to synchronize List, Set and Map elements?

Type	How to Synchronize?
List	• Collections.synchronizedList() method. • Using concurrent List
Set	• Collections.synchronizedSet() method. • Using concurrent Set
Map	• Collections.synchronizedMap() method.

12. What do you understand by fail-fast?

Fail Fast and Fail-Safe Systems:

- The Fail Fast system is a system that shuts down immediately after an error is reported. All the operations will be aborted instantly in it.
- The Fail Safe is a system that continues to operate even after an error or fail has occurred. These systems do not abort the operations instantly; instead, they will try to hide the errors and will try to avoid failures as much as possible.

Fail Fast Iterator:

The iterator in Java is used to traverse over a collection's objects. The collections return two types of iterators, either it will be Fail Fast or Fail Safe.

The Fail Fast iterators immediately throw ConcurrentModificationException in case of structural modification of the collection. Structural modification means adding, removing, updating the value of

an element in a data collection while another thread is iterating over that collection. Some examples of Fail Fast iterator are iterator on ArrayList, HashMap collection classes.

How it Works:

The Fail Fast iterator uses an internal flag called **modCount** to know the status of the collection, whether the collection is structurally modified or not. The modCount flag is updated each time a collection is modified; it checks the next value; if it finds, then the modCount will be modified after this iterator has been created. It will throw ConcurrentModificationException.

13. What are the differences between Array and ArrayList?

Base	Array	ArrayList
Dimensionality	It can be single-dimensional or multidimensional	It can only be single-dimensional
Traversing Elements	For and for each generally is used for iterating over arrays	Here iterator is used to traverse riverArrayList
Length	length keyword can give the total size of the array.	size() method is used to compute the size of ArrayList.
Size	It is static and of fixed length	It is dynamic and can be increased or decreased in size when required.
Speed	It is faster as above we see it of fixed size	It is relatively slower because of its dynamic nature
Primitive Datatype Storage	Primitive data types can be stored directly unlikely objects	Primitive data types are not directly added unlikely arrays, they are added indirectly with help of autoboxing and unboxing
Generics	They can not be added here hence type unsafe	They can be added here hence makingArrayList type-safe.
Adding Elements	Assignment operator only serves the purpose	Here a special method is used known as add() method

14. How to remove duplicates from an ArrayList?

There are three ways which can be used to remove a duplicate from an ArrayList:

A. Using Iterator

Approach:

1. Get the ArrayList with duplicate values.
2. Create another ArrayList.
3. Traverse through the first arraylist and store the first appearance of each element into the second arraylist using contains() method.
4. The second ArrayList contains the elements with duplicates removed.

B. Using LinkedHashSet

A better way (both time complexity and ease of implementation wise) is to remove duplicates from an ArrayList is to convert it into a Set that does not allow duplicates. Hence

LinkedHashSet is the best option available as this do not allows duplicates as well it preserves the insertion order.

Approach:

1. Get the ArrayList with duplicate values.
2. Create a LinkedHashSet from this ArrayList. This will remove the duplicates
3. Convert this LinkedHashSet back to Arraylist.
4. The second ArrayList contains the elements with duplicates removed.

C. Using Java 8 stream.distinct()

You can use the distinct() method from the Stream API. The distinct() method return a new Stream without duplicates elements based on the result returned by equals() method, which can be used for further processing. The actual processing of Stream pipeline starts only after calling terminal methods like forEach() or collect().

Approach:

1. Get the ArrayList with duplicate values.
2. Create a new List from this ArrayList.
3. Using stream().distinct() method which return distinct object stream.
4. convert this object stream into List.

15. Write a Java program to copy one ArrayList to another.

```
import java.util.ArrayList;

import java.util.Iterator;



public class CopyArrayList {

	public static void main(String[] args) {

		ArrayList <Integer> first = new ArrayList<>();

		first.add(10);

		first.add(20);

		first.add(30);

		first.add(40);

		System.out.println("--------First List---------");

		for (Integer integer : first) {

			System.out.println(integer);

		}
```

```
            ArrayList<Integer> second = new ArrayList<Integer>(first);
            System.out.println("---------Second List-------");
            for (Integer integer : second) {
                  System.out.println(integer);
            }
            second.set(2, 32);
            System.out.println("---------After Modification-------");
            System.out.println("--------First List---------");
            for (Integer integer : first) {
                  System.out.println(integer);
            }
            System.out.println("--------Second List---------");
            for (Integer integer : second) {
                  System.out.println(integer);
            }
      }
}
```

16. Write a program to swap two elements of an ArrayList.

```
import java.util.ArrayList;
import java.util.Collections;

public class SwapTwoElementsArrayList {
   public static void main(String[] args) {
          ArrayList<String> ArrList = new ArrayList<String>();
          // add the values in Array List
        ArrList.add("Item 1");
        ArrList.add("Item 2");
        ArrList.add("Item 3");
        ArrList.add("Item 4");
```

```
        ArrList.add("Item 5");
        // display Array List before swap
        System.out.println("Before Swap the ArrayList ");
        System.out.println(ArrList);
        // Swapping the elements at 1 and 2 indices
        Collections.swap(ArrList, 1, 2);
         // display Array List after swap
        System.out.println("After Swap the ArrayList");
        System.out.println(ArrList);
    }
}
```

17. Write a program to iterate through all the elements in a LinkedList starting at a specified position.

```
import java.util.Iterator;
import java.util.LinkedList;
public class IterateFromSpecifiedIndexLinkedList {
    public static void main(String[] args) {
            LinkedList<String> l_list = new LinkedList<String>();
            l_list.add("Red");
            l_list.add("Green");
            l_list.add("Black");
            l_list.add("White");
            l_list.add("Pink");
            // set Iterator at specified index
            Iterator p = l_list.listIterator(4);
            // print list from set position
            while (p.hasNext()) {
                    System.out.println(p.next());
            }
```

```
    }

}
```

18. Write a program to get the first and last occurrence of a specified element in a LinkedList.

```
import java.util.LinkedList;

public class FirstAndLastOccuranceLinkedList {

    public static void main(String[] args) {

            LinkedList<String> list = new LinkedList<String>();

            // Add elements

            list.add("AA");

            list.add("BB");

            list.add("CC");

            list.add("AA");

            list.add("DD");

            list.add("AA");

            list.add("EE");

            // Display LinkedList elements

            System.out.println("LinkedList elements: " + list);

            System.out.println("\n-----First occurance of AA---------");

            // +1 to emulate the actual position in the list

            System.out.println(list.indexOf("AA") + 1);



            System.out.println("-----Last occurance of AA---------");

            System.out.println(list.lastIndexOf("AA") + 1);

    }

}
```

19. Write a program to retrieve but does not remove the first element of an LinkedList.

```
import java.util.LinkedList;



public class GetFirstElementLinkedList {

   public static void main(String[] args) {

          LinkedList<String> list = new LinkedList<String>();



          // Add elements

          list.add("AA");

          list.add("BB");

          list.add("CC");

          list.add("AA");

          list.add("DD");

          list.add("AA");

          list.add("EE");

          System.out.println("First Element : " + list.getFirst());

          // to demonstrate that this doesn't affect the LinkedList

          System.out.println(list);

   }

}
```

20. Write a Java Program to convert LinkedList to ArrayList

```
import java.util.ArrayList;

import java.util.LinkedList;

public class LinkedListToArrayList {

   public static void main(String[] args) {

          LinkedList<String> linkedlist = new LinkedList<String>();

          linkedlist.add("Harry");

          linkedlist.add("Jack");
```

```
            linkedlist.add("Tim");

            linkedlist.add("Rick");

            linkedlist.add("Rock");


            ArrayList<String> list = new ArrayList<String>(linkedlist);


            for (String str : list) {

                  System.out.println(str);

            }

      }

}
```