Output:

```
Count is : 100
```

Question 94: What is Collection Framework?

Answer: Collection framework represents an architecture to store and manipulate a group of objects. All the classes and interfaces of this framework are present in java.util package.

Some points:

- Iterable interface is the root interface for all collection classes, it has one abstract method iterator()
- Collection interface extends the Iterable interface

Question 95: What is Collections?

Answer: Collections is a utility class present in java.util package that we can use while using collections like List, Set, and Map etc.

Some commonly used methods are Collections.sort(), Collections.unmodifiableList() etc.

Question 96: What is ArrayList?

Answer: ArrayList is a resizable-array implementation of List Interface. When we create an object of ArrayList then a contiguous block of memory is allocated to hold its elements. As it is a contiguous block, all the elements address is already known, that is how ArrayList allows random access.

Some points about ArrayList class:

- ArrayList class maintains insertion order
- ArrayList class can contain duplicate elements
- ArrayList class allows random access to its elements as it works on index basis
- ArrayList class is not synchronized
- You can add any number of null elements in the ArrayList class

Time complexity of ArrayList's get(), add(), remove() operations:

get(): constant time

add(): here the new element can be added at the first, middle or last positions, if you are using add(element), then it will append element at the last position and will take O(1), provided that the arrayList is not full otherwise it will create a new arrayList of one and a half times the size of previous arrayList and copy all arrayList elements to this new arrayList, making it O(n).

If the element is added in the middle or at any specific index, let's say at index 2, then a space needs to be made to insert this new element by shifting all the elements one position to its right, making it O(n).

add() operation runs in amortized constant time.

remove(): it is also same as add(), if you want to remove element from a specific index, then all elements to its right needs to be shifted one position to their left, making it O(n) but if element needs to be removed from the last, then it will take O(1).

Question 97: What is default size of ArrayList?

Answer: 10

```
/**
* Default initial capacity.
*/
private static final int DEFAULT_CAPACITY = 10;
```

Question 98: Which data structure is used internally in an ArrayList?

Answer: Internally, ArrayList uses Object[]

```
/**
* The array buffer into which the elements of the ArrayList are stored.
* The capacity of the ArrayList is the length of this array buffer. Any
* empty ArrayList with elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA
* will be expanded to DEFAULT_CAPACITY when the first element is added.
*/
transient Object[] elementData;
```

Question 99: How add() method works internally or How the ArrayList grows at runtime

Answer: this is what happens when we create an ArrayList object using its default constructor,

```
/**
* Constructs an empty list with an initial capacity of ten.
*/
public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}
```

Here, elementData is a transient variable and DEFAULTCAPACITY_EMPTY_ELEMENTDATA is an empty Object[] array:

```
transient Object[] elementData;

private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
```

Now, let's see Javadoc of add() method

```
/**
* Appends the specified element to the end of this list.
*
* @param e element to be appended to this list
* @return <tt>true</tt> (as specified by {@link Collection#add})
*/
public boolean add(E e) {
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
```

Here size is a private variable

```
/**
* The size of the ArrayList (the number of elements it contains).
*
* @serial
*/
private int size;
```

The default value of size will be 0, so call to ensureCapacityInternal() will have value 1, now let's see ensureCapacityInternal() Javadoc:

```
private void ensureCapacityInternal(int minCapacity) {
    ensureExplicitCapacity(calculateCapacity(elementData,minCapacity));
}
```

Here minCapacity is holding value 1, calculateCapacity() method is:

```
private static int calculateCapacity(Object[] elementData, int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        return Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    return minCapacity;
```

```
}
```

Now, as both elementData and DEFAULTCAPACITY_EMPTY_ELEMENTDATA are same (see the default ArrayList constructor above), if condition will be true and then Math.max(10,1) will return 10 from calculateCapacity() method

Now, 10 will be passed to ensureExplicitCapacity()

```
private void ensureCapacityInternal(int minCapacity) {
    ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));
}

private void ensureExplicitCapacity(int minCapacity) {
    modCount++;

    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
```

modCount is used when we are iterating over ArrayList using Iterator or ListIterator, here minCapacity is 10 and elementData.length will be 0, so if condition will be satisfied and grow() method will be called with value 10:

```
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

Here, oldCapacity will be 0, and newCapacity will also be 0, so the first if condition will be satisfied because (0-10 < 0), so newCapacity will be minCapacity i.e. 10, the second if condition will not be satisfied as MAX_ARRAY_SIZE is a very huge number,

```
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
```

So, the ensureCapacityInternal() of add() will be executed :

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
```

Element e will be added to object array elementData at index 0 and size will be incremented and finally add() method will return true, now this same process will repeat.

So, we can say before adding the element in arrayList, first it is ensured that the array can hold the element, if not the capacity will be increased and for this, grow() method is called. Suppose you are trying to add 11^{th} element in the list, then grow() method will be called and the statement int newCapacity = oldCapacity + (oldCapacity >> 1); will make the new capacity to be one and a half times(50% increase) the size of list.

Let's make a simple code to understand this line:

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

        int oldCapacity = 10;

        int newCapacity = oldCapacity + (oldCapacity >> 1);

        System.out.println(newCapacity);
    }
}
```

Output:

```
15
```

Question 100: How to make an ArrayList as Immutable

Answer: This is also a common interview question nowadays. If your answer is making the list as "final" then see the code below:

Program 1:

```java
public class TestArrayList {
    public static void main(String[] args) {
        final List<String> list = new ArrayList<>();

        list.add("John");
        list.add("Mike");
        list.add("Lisa");

        System.out.println(list);
    }
}
```

Output:

```
[John, Mike, Lisa]
```

Although, we have made the list as final but still we are able to add elements into it, remember applying final keyword to a reference variable ensures that it will not be referenced again meaning you cannot give a new reference to list variable:

```java
public class TestArrayList {
    public static void main(String[] args) {
        final List<String> list = new ArrayList<>();

        list.add("John");
        list.add("Mike");
        list.add("Lisa");

        list = new ArrayList<>();
        System.out.println(list);
    }
}
```

Errors (1 item)

The final local variable list cannot be assigned. It must be blank and not using a compound assignment

So, to make the list as unmodifiable, there is a method unmodifiableList() in Collections utility class,

Program 2:

```java
public class TestArrayList {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("John");
        list.add("Mike");
        list.add("Lisa");
        System.out.println(list);

        list = Collections.unmodifiableList(list);
        list.add("Jack");
        System.out.println(list);
    }
}
```

Output:

```
[John, Mike, Lisa]
Exception in thread "main" java.lang.UnsupportedOperationException
        at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)
        at com.collections.demo.TestArrayList.main(TestArrayList.java:16)
```

Here, if you assign Collections.*unmodifiableList*(list); to a new reference then you will be able to change the original list which will in turn change the new list also, see below:

Program 3:

```
public class TestArrayList {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("John");
        list.add("Mike");
        list.add("Lisa");

        List<String> newList = Collections.unmodifiableList(list);
        list.add("Jack");
        System.out.println(newList);
    }
}
```

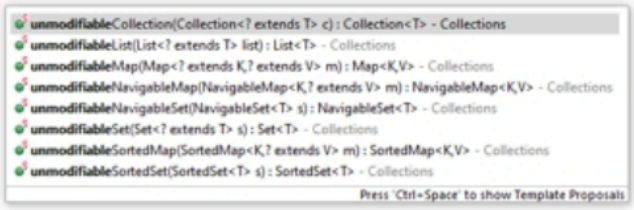
Output:

```
[John, Mike, Lisa, Jack]
```

Guava library also provides certain ways to make immutable list and Java 9 has List.of() method.

There are other utility methods also, to make unmodifiable collections:

```
Collections.unmodifiable
```



Question 101: What is LinkedList?

Answer: Java LinkedList class is an implementation of linked list data structure and it uses a doubly linked list to store the elements. In Java LinkedList, elements are not stored in contiguous locations, they are stored at any available space in memory and they are linked with each other using pointers and addresses.

As Java LinkedList internally uses doubly linked list, so LinkedList class represents its elements as Nodes. A Node is divided into 3 parts:

Previous, Data, Next

Where Previous points to the previous Node in the list, Next points to the next Node in the list and Data is the actual data.

Some points about LinkedList class:

- LinkedList class maintains insertion order
- LinkedList class can contain duplicate elements
- LinkedList class is not synchronized
- LinkedList class can be used as list, stack or queue
- You can add any number of null elements in LinkedList

Time complexity of LinkedList's get(), add() and remove():

get(): As LinkedList does not store its elements in contiguous block of memory, random access is not supported here, elements can be accessed in sequential order only, so get() operation in LinkedList is O(n).

add() and ***remove():*** Both add and remove operations in LinkedList is O(1), because no elements shifting is needed, just pointer modification is done (although remember getting to the index where you want to add/remove will still be O(n)).

Here, I am showing some portions of LinkedList Javadoc's:

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
```

```
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
{
    transient int size = 0;

    /**
     * Pointer to first node.
     * Invariant: (first == null && last == null) ||
     *        (first.prev == null && first.item != null)
     */
    transient Node<E> first;

    /**
     * Pointer to last node.
     * Invariant: (first == null && last == null) ||
     *        (last.next == null && last.item != null)
     */
    transient Node<E> last;

    /**
     * Constructs an empty list.
     */
    public LinkedList() {
    }
```

We can see that the LinkedList class implements List and Deque interfaces. There are first and last Node references also.

Let's see the add() method:

```
public boolean add(E e) {
    linkLast(e);
    return true;
}

void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<>(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode;
    else
        l.next = newNode;
    size++;
    modCount++;
}
```

Here, in linkLast() method, Node class is used, let' see that:

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

Here, we can see that Node class has 3 fields: item, prev and next.

Question 102: When to use ArrayList / LinkedList