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:

```
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:

```
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:

```java
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);
    }
}
```

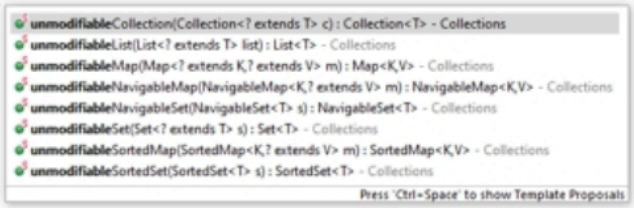
Output:

```
[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

Answer: When you have a requirement in which you will be doing a lot of add or remove operations near the middle of list, then prefer LinkedList, and when you have a requirement, where the frequently used operation is searching an element from the list, then prefer ArrayList as it allows random access.

Question 103: What is HashMap?

Answer: HashMap class implements the Map interface and it stores data in key, value pairs. HashMap provides constant time performance for its get() and put() operations, assuming the equals and hashcode method has been implemented properly, so that elements can be distributed correctly among the buckets.

Some points to remember:

- Keys should be unique in HashMap, if you try to insert the duplicate key, then it will override the corresponding key's value
- HashMap may have one null key and multiple null values
- HashMap does not guarantee the insertion order (if you want to maintain the insertion order, use LinkedHashMap class)
- HashMap is not synchronized
- HashMap uses an inner class Node<K, V> for storing map entries
- Hashmap has a default initial capacity of 16, which means it has 16 buckets or bins to store map entries, each bucket is a singly linked list. The default load factor in HashMap is 0.75
- Load factor is that threshold value which when crossed will double the hashmap's capacity i.e. when you add 13^{th} element in hashmap, the capacity will increase from 16 to 32

Question 104: Explain the internal working of put() and get() methods of HashMap class and discuss HashMap collisions

Answer: If you are giving a Core java interview, then you must prepare for this question, as you will most likely be asked about this. So, let's get right into it:

put() method internal working:

When you call map.put(key,value), the below things happens:

- Key's hashCode() method is called
- Hashmap has an internal hash function which takes the key's hashCode and it calculates the bucket index
- If there is no element present at that bucket index, our <key, value> pair along with hash is stored at that bucket
- But if there is an element present at the bucket index, then key's hashCode is used to check whether this key is already present with the same hashCode or not.

If there is key with same hashCode, then equals method is used on the key. If equals method returns true, then the key's previous value is replaced with the new value otherwise a new entry is appended to the linked list.

get() method internal working:

When you call map.get(key), the below things happen:

- Key's hashCode() method is called
- Hash function uses this hashCode to calculate the index, just like in put method
- Now the key of element stored in bucket is compared with the passed key using equals() method, if both are equals, value is returned otherwise the next element is checked if it exists.

See HashMap's Javadoc:

Default capacity:

```
/**
* The default initial capacity - MUST be a power of two.
*/
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
```

Load factor:

```
/**
* The load factor used when none specified in constructor.
*/
static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

Node class:

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;

    Node(int hash, K key, V value, Node<K,V> next) {
        this.hash = hash;
        this.key = key;
        this.value = value;
        this.next = next;
    }
```

Internal Hash function:

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

Internal Data structure used by HashMap to hold buckets:

```
transient Node<K,V>[] table;
```

HashMap's default constructor:

```
/**
* Constructs an empty <tt>HashMap</tt> with the default initial capacity
* (16) and the default load factor (0.75).
*/
public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
}
```

So, to conclude, Hashmap internally uses an array of Nodes named as table where each Node contains the calculated hash value, the key-value pair and the address to the next node.

HashMap collisions: it is possible that multiple keys will make the hash function generate the same index, this is called a collision. It happens because of poor hashcode method implementation.

One collision handling technique is called Chaining. Since every element in the array is a linked list, the keys which have the same hash function will be appended to the linked list.

Performance improvement in Java 8: It is possible that due to multiple collisions, the linked list size has become very large, and as we know, searching in a linked list is O(n), it will impact the constant time performance of hashmap's get() method. So, in Java 8, if the linked list size becomes more than

8, the linked list is converted to a binary search tree which will give a better time complexity of O(log n).

```
/**
* The bin count threshold for using a tree rather than list for a
* bin.  Bins are converted to trees when adding an element to a
* bin with at least this many nodes. The value must be greater
* than 2 and should be at least 8 to mesh with assumptions in
* tree removal about conversion back to plain bins upon
* shrinkage.
*/
static final int TREEIFY_THRESHOLD = 8;
```

Program showing the default capacity:

```
public class TestHashMap {
    public static void main(String[] args) throws Exception {
        HashMap<String, String> map = new HashMap<>();
        map.put("name", "Mike");

        Field tableField = HashMap.class.getDeclaredField("table");
        tableField.setAccessible(true);
        Object[] table = (Object[]) tableField.get(map);
        System.out.print("hashmap capacity: ");
        System.out.print(table == null ? 0 : table.length);
        System.out.println("\nhashmap size:" + map.size());
    }
}
```

Output:

```
hashmap capacity: 16
hashmap size:1
```

Program showing that hashmap's capacity gets doubled after load factor's threshold value breaches:

```
package com.hashmap.demo;

import java.lang.reflect.Field;
import java.util.HashMap;

class Employee {
    private int age;

    public Employee(int age) {
        this.age = age;
    }
}
```

```
public class TestHashMap {
    public static void main(String[] args) throws Exception {
        HashMap<Employee, String> map = new HashMap<>();

        for(int i=1;i<13;i++) {
            map.put(new Employee(i), "Hello " + i);
        }

        Field tableField = HashMap.class.getDeclaredField("table");
        tableField.setAccessible(true);
        Object[] table = (Object[]) tableField.get(map);
        System.out.print("hashmap capacity: ");
        System.out.print(table == null ? 0 : table.length);
        System.out.println("\nhashmap size:" + map.size());
    }
}
```

Output:

```
hashmap capacity: 16
hashmap size:12
```

Change the for loop condition from i<13 to i<=13, see below:

```
package com.hashmap.demo;

import java.lang.reflect.Field;
import java.util.HashMap;

class Employee {
    private int age;

    public Employee(int age) {
        this.age = age;
    }
}
public class TestHashMap {
    public static void main(String[] args) throws Exception {
        HashMap<Employee, String> map = new HashMap<>();

        for(int i=1;i<=13;i++) {
            map.put(new Employee(i), "Hello " + i);
        }

        Field tableField = HashMap.class.getDeclaredField("table");
        tableField.setAccessible(true);
        Object[] table = (Object[]) tableField.get(map);
        System.out.print("hashmap capacity: ");
        System.out.print(table == null ? 0 : table.length);
        System.out.println("\nhashmap size:" + map.size());
    }
}
```

Output:

```
hashmap capacity: 32
hashmap size:13
```

Question 105: equals and hashCode method scenarios in HashMap when the key is a custom class

Answer: equals and hashCode methods are called when we store and retrieve values from hashmap. So, when the interviewer asks this question, it is mostly asked with an example, where the hashmap's key is a custom class and you are given some situations where either equals() is implemented or hashCode() is implemented, sometimes properly, sometimes not. We will discuss all combinations with programs below so you can give the correct answer in any situation.

Before we continue, just remember if in your custom class, you are not implementing equals() and hashCode(), then the Object class equals() and hashCode() will be called, and also remember about the contract between these 2 methods. It says when 2 objects are equal according to equals() method, then their hashCode must be same, reverse may not be true.

Scenario 1: when custom class does not implement both equals and hashCode methods

```
package com.hashmap.demo;

public class Employee {
    private String name;
    private int age;

    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Here, Employee class has not given equals() and hashCode() method implementation, so Object's class equals() and hashCode() methods will be used when we use this Employee class as hashmap's key, and remember, equals() method of Object class compares the reference.

TestHashMap.java:

```java
package com.hashmap.demo;

import java.util.HashMap;
import java.util.Map;

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

        Map<Employee, Integer> map = new HashMap<>();

        Employee e1 = new Employee("Mike", 15);
        Employee e2 = new Employee("Mike", 15);
        Employee e3 = new Employee("John", 20);
        Employee e4 = e3;

        System.out.println("e1 hashcode: " + e1.hashCode());
        System.out.println("e2 hashcode: " + e2.hashCode());
        System.out.println("e3 hashcode: " + e3.hashCode());
        System.out.println("e4 hashcode: " + e4.hashCode());

        System.out.println("e1 equals e2: " + e1.equals(e2));
        System.out.println("e3 equals e4: " + e3.equals(e4));

        map.put(e1, 100);
        map.put(e2, 200);
        map.put(e3, 300);
        map.put(e4, 400);

        System.out.println(map.get(e1));
        System.out.println(map.get(e2));
        System.out.println(map.get(e3));
        System.out.println(map.get(e4));
        System.out.println("hashmap size: " + map.size());
    }
}
```

Can you predict the output of this one?

Output:

```
e1 hashcode: 366712642
e2 hashcode: 1829164700
e3 hashcode: 2018699554
e4 hashcode: 2018699554
e1 equals e2: false
e3 equals e4: true
100
200
400
400
hashmap size: 3
```

Here, Employee objects e1 and e2 are same but they are both inserted in the HashMap because both are created using new keyword and holding a different reference, and as the Object's equals() method checks reference, they both are unique.

And as for objects e3 and e4, they both are pointing to same reference (e4 = e3), so they are equal according to Object's equals() method hence the value of e3 which was 300 gets replaced with the value 400 of the same key e4, and finally size of HashMap is 3.

Scenario 2: when only equals() method is implemented by Employee class

```
package com.hashmap.demo;

public class Employee {

    private String name;
    private int age;

    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }
```

```java
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Employee other = (Employee) obj;
        if (age != other.age)
            return false;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        return true;
    }
}
```

Now, can you predict the output of our TestHashMap class?

Let's see the output:

```
e1 hashcode: 366712642
e2 hashcode: 1829164700
e3 hashcode: 2018699554
e4 hashcode: 2018699554
e1 equals e2: true
e3 equals e4: true
100
200
400
400
hashmap size: 3
```

Well, nothing's changed here. Because even though e1 and e2 are equal according to our newly implemented equals() method, they still have different hashCode as the Object's class hashCode() is used. So the equals and hashCode contract is not followed and both e1, e2 got inserted in HashMap.

Scenario 3: when only hashCode() method is implemented:

```java
package com.hashmap.demo;

public class Employee {
    private String name;
    private int age;
    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + age;
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Employee other = (Employee) obj;
        if (age != other.age)
            return false;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        return true;
    }
}
```

Output:

```
e1 hashcode: 2399656
e2 hashcode: 2399656
e3 hashcode: 2316120
e4 hashcode: 2316120
e1 equals e2: true
e3 equals e4: true
200
200
400
400
hashmap size: 2
```

Here, both e1 and e2 are equals as we are comparing the contents of them in our equals() method, so their hashCodes must be same, which they are. So value of e1 which was 100 got replaced by 200, and size of hashmap is 2.

You can be asked to write the equals() and hashCode() methods implementation by hand also, so you should pay attention to how these are implemented.

Question 106: How to make a HashMap synchronized?

Answer: Collections.synchronizedMap(map);

Question 107: What is Concurrent HashMap?

Answer: ConcurrentHashMap class provides concurrent access to the map, this class is very similar to HashTable, except that ConcurrentHashMap provides better concurrency than HashTable or even synchronizedMap.

Some points to remember:

- ConcurrentHashMap is internally divided into segments, by default size is 16 that means, at max 16 threads can work at a time
- Unlike HashTable, the entire map is not locked while reading/writing from the map
- In ConcurrentHashMap, concurrent threads can read the value without locking
- For adding or updating the map, the lock is obtained on segment level, that means each thread can work on each segment during high concurrency
- Concurrency level defines a number, which is an estimated number of threads concurrently accessing the map
- ConcurrentHashMap does not allow null keys or null values
- put() method acquires lock on the segment
- get() method returns the most recently updated value
- iterators returned by ConcurrentHashMap are fail-safe and never throw ConcurrentModificationException

Question 108: What is HashSet class and how it works internally?

Answer: HashSet is a class in Java that implements the Set Interface and it allows us to have the unique elements only. HashSet class does not maintain the insertion order of elements, if you want to maintain the insertion order, then you can use LinkedHashSet.

Internal implementation of HashSet:

HashSet internally uses HashMap and as we know the keys are unique in hashmap, the value passed in the add() method of HashSet is stored as the key of hashmap, that is how Set maintains the unique elements.

```
/**
* Constructs a new, empty set; the backing <tt>HashMap</tt> instance has
* default initial capacity (16) and load factor (0.75).
*/
public HashSet() {
    map = new HashMap<>();
}

private transient HashMap<E,Object> map;
```

Let's see add() method's Javadoc:

```
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}

// Dummy value to associate with an Object in the backing Map
private static final Object PRESENT = new Object();
```

So, when we call hashSet.add(element) method then

- map.put() is called where key is the element and value is the dummy value (the map.put() method internal working has already been discussed above)
- if value is added in the map then put method will return null which will be compared with null, hence returning true from hashSet.add() method indicating the element is added
- however if the element is already present in the map, then the value associated with the element will be returned which in turn will be compared with null, returning false from hashSet.add() method

set.contains() method:

```
public boolean contains(Object o) {
    return map.containsKey(o);
}
```

The passed object is given to map.containsKey() method, as the HashSet's values are stored as the keys of internal map.

NOTE: If you are adding a custom class object inside the HashSet, do follow equals and hashCode contract. You can be asked the equals and hashCode scenarios questions, just like we discussed in HashMap (Question 105).

Question 109: Explain Java's TreeMap

Answer: TreeMap class is one of the implementation of Map interface.

Some points to remember:

- TreeMap entries are sorted based on the natural ordering of its keys. This means if we are using a custom class as the key, we have to make sure that the custom class is implementing Comparable interface
- TreeMap class also provides a constructor which takes a Comparator object, this should be used when we want to do a custom sorting
- TreeMap provides guaranteed log(n) time complexity for the methods such as containsKey(), get(), put() and remove()
- TreeMap iterator is fail-fast in nature, so any concurrent modification will result in ConcurrentModificationException
- TreeMap does not allow null keys but it allows multiple null values
- TreeMap is not synchronized, so it is not thread-safe. We can make it thread-safe by using utility method, Collections.synchronizedSortedMap(treeMap)
- TreeMap internally uses Red-Black tree based NavigableMap implementation.

Red-Black tree algorithm has the following properties:

- Color of every node in the tree is either red or black.
- Root node must be Black in color.
- Red node cannot have a red color neighbor node.
- All paths from root node to the null should consist the same number of black nodes

Program 1: Using Wrapper class as key

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

        TreeMap<Integer, String> map = new TreeMap<>();

        map.put(4, "Mike");
        map.put(1, "John");
        map.put(3, "Jack");
        map.put(2, "Lisa");

        map.forEach((k,v) -> System.out.println(k + ":" + v));
    }
}
```

Output:

```
1:John
2:Lisa
3:Jack
4:Mike
```

Here, Integer class already implements Comparable interface, so the keys are sorted based on the Integer's natural sorting order (ascending order).

Let's see, when key is a custom class:

Program 2:

```
class Employee {
    String name;
    int age;
    Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
public class TestTreeMap {
    public static void main(String[] args) {

        TreeMap<Employee, Integer> map = new TreeMap<>();

        map.put(new Employee("Mike", 20), 100);
        map.put(new Employee("John", 10), 500);
        map.put(new Employee("Ryan", 15), 200);
        map.put(new Employee("Lisa", 20), 400);

        map.forEach((k,v) -> System.out.println(k + ":" + v));
    }
}
```

Output:

```
Exception in thread "main" java.lang.ClassCastException: com.treemap.demo.Employee cannot be cast to java.lang.Comparable
        at java.util.TreeMap.compare(Unknown Source)
        at java.util.TreeMap.put(Unknown Source)
        at com.treemap.demo.TestTreeMap.main(TestTreeMap.java:19)
```

We get ClassCastException at runtime. Now, let's implement Comparable interface in Employee class and provide implementation of its compareTo() method:

Program 3: