Q. what is the Set and Why Use Set Collection in java?

Basically, **Set** is a type of collection that does not allow duplicate elements. That means an element can only exist once in a Set. It models the set abstraction in mathematics.

Characteristics of a Set collection:

The following characteristics differentiate a Set collection from others in the Java Collections framework:

- Duplicate elements are not allowed.
- Elements are not stored in order. That means you cannot expect elements sorted in any order when iterating over elements of a Set.

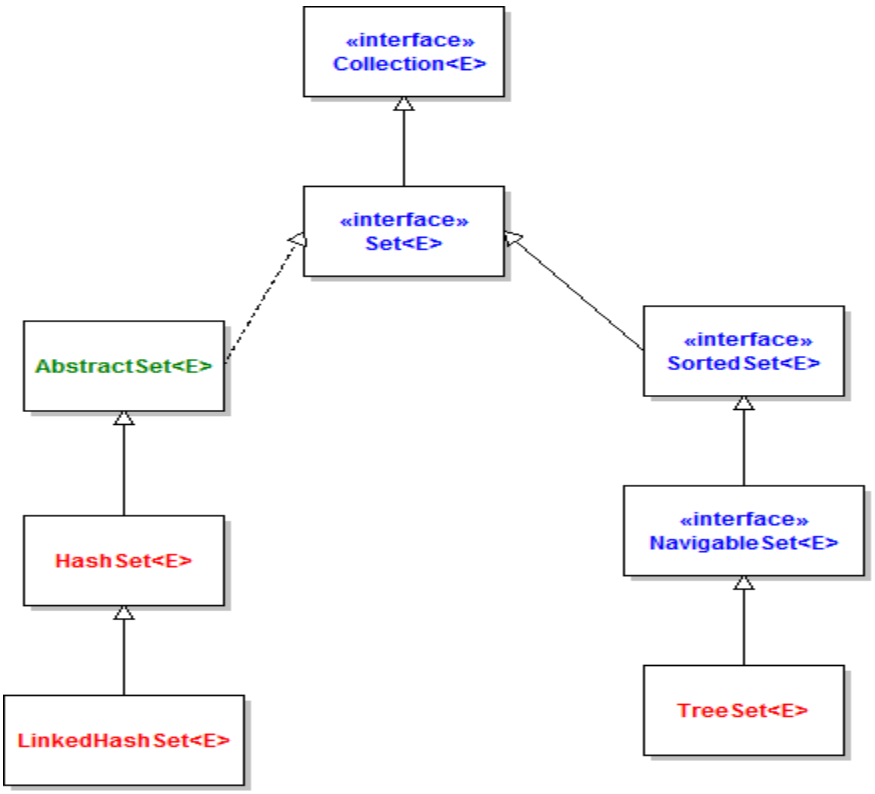
Why and When Use Sets?

Based on the characteristics, consider using a Set collection when:

- You want to store elements distinctly without duplication, or unique elements.
- You don't care about the order of elements.

For example, you can use a Set to store unique integer numbers; you can use a Set to store cards randomly in a card game; you can use a Set to store numbers in random order, etc.

The Java Collections Framework provides three major implementations of the Set interface: HashSet, LinkedHashSet and TreeSet. The Set API is described in the following diagram:



Let's look at the characteristics of each implementation in details:

- **HashSet**: is the best-performing implementation and is a widely-used Set implementation. It represents the core characteristics of sets: no duplication and unordered.
- **LinkedHashSet**: This implementation orders its elements based on insertion order. So consider using a LinkedHashSet when you want to store unique elements in order.
- **TreeSet**: This implementation orders its elements based on their values, either by their natural ordering, or by a Comparator provided at creation time.

Therefore, besides the uniqueness of elements that a Set guarantees, consider using HashSet when ordering does not matter; using LinkedHashSet when you want to order elements by their insertion order; using TreeSet when you want to order elements by their values.

Q. What is HashSet? What are the properties of HashSet object?

1. HashSet ensures uniqueness, In other words , every object in the HashSet presents only once
2. Unlike LinkedList, HashSet does not store the objects in orderly manner. It is possible that the object which we added first in the HashSet , may appear last in the output .
3. It offers constant time performance for basic operations like add, remove , contains and size assuming the hash function disperses the element properly among the buckets.
4. HashSet implementation is not synchronized.

* If multiple threads access a hash set concurrently, and at least one of the threads modifies the set, it must be synchronized externally.

*This is typically accomplished by synchronizing on some object that naturally encapsulates the set.

Q. 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.

Q. What is the difference between HashSet and TreeSet?

Ordering: HashSet stores the object in random order . There is no guarantee that the element we inserted first in the HashSet will be printed first in the output .

For example

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

```
   public static void main(String[] args) {
      HashSet<String>  obj1= new HashSet<String>();
      obj1.add("Alive");
      obj1.add("is");
      obj1.add("Awesome");
      System.out.println(obj1);
   }
}
```

Elements are sorted according to the natural ordering of its elements in TreeSet. If the objects can not
be sorted in natural order than use compareTo() method to sort the elements of TreeSet object

```
import java.util.TreeSet;
public class TreeSetExample {
   public static void main(String[] args) {
      TreeSet<String>  obj1= new TreeSet<String>();
      obj1.add("Alive");
      obj1.add("is");
      obj1.add("Awesome");
      System.out.println(obj1);
   }
}
```

2. Null value: HashSet can store null object while TreeSet does not allow null object. If one try to store null object in TreeSet object , it will throw Null Pointer Exception.

3. Performance : HashSet take constant time performance for the basic operations like add, remove contains and size.While TreeSet guarantees log(n) time cost for the basic operations (add,remove,contains).

4. Speed: HashSet is much faster than TreeSet,as performance time of HashSet is constant against the log time of TreeSet for most operations (add,remove ,contains and size) . Iteration performance of HashSet mainly depends on the load factor and initial capacity parameters.

5. Internal implementation : As we have already discussed How hashset internally works in java thus, in one line HashSet are internally backed by hashmap. While TreeSet is backed by a Navigable TreeMap.
6. Functionality : TreeSet is rich in functionality as compare to HashSet. Functions like pollFirst(),pollLast(),first(),last(),ceiling(),lower() etc. makes TreeSet easier to use than HashSet.
7. Comparision : HashSet uses equals() method for comparison in java while TreeSet uses compareTo() method for maintaining ordering .

Q. What Are The Similarities Between HashSet and TreeSet
1. Unique Elements : Since HashSet and TreeSet both implements Set interface . Both are allowed to store only unique elements in their objects. Thus there can never be any duplicate elements inside the HashSet and TreeSet objects.

2. Not Thread Safe : HashSet and TreeSet both are not synchronized or not thread safe.HashSet and TreeSet, both implementations are not synchronized. If multiple threads access a hash set/ tree set concurrently, and at least one of the threads modifies the set, it must be synchronized externally.

3. Clone() method copy technique: Both HashSet and TreeSet uses shallow copy technique to create a clone of their objects .

4. Fail-fast Iterators : The iterators returned by this class's method are fail-fast: if the set is modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Q. When to prefer TreeSet over HashSet
1. Sorted unique elements are required instead of unique elements. The sorted list given by TreeSet is always in ascending order.

2. TreeSet has greater locality than HashSet.

If two entries are nearby in the order , then TreeSet places them near each other in data structure and hence in memory, while HashSet spreads the entries all over memory regardless of the keys they are associated to.

As we know Data reads from the hard drive takes much more latency time than data read from the cache or memory. In case data needs to be read from hard drive than prefer TreeSet as it has greater locality than HashSet.

3. TreeSet uses Red- Black tree algorithm underneath to sort out the elements. When one need to perform read/write operations frequently , then TreeSet is a good choice.

What is the LinkedHashSet?

LinkedHashSet collection can store the unique data in collection and manage the data sequence as per the user sequence.
The LinkedHashSet class of the Java collections framework provides functionalities of both the hashtable and the linked list data structure. However, linked hash sets maintain a doubly-linked list internally for all of its elements. The linked list defines the order in which elements are inserted in hash tables.
In order to create a linked hash set, we must import
the java.util.LinkedHashSet package first.
Once we import the package, here is how we can create linked hash sets in Java.

```
// LinkedHashSet with 8 capacity and 0.75 load factor
LinkedHashSet numbers = new LinkedHashSet(8, 0.75);
```

Notice, the part new LinkedHashSet(8, 0.75). Here, the first parameter is **capacity** and the second parameter is **loadFactor**.

- **capacity** - The capacity of this hash set is 8. Meaning, it can store 8 elements.

- **loadFactor** - The load factor of this hash set is 0.6. This means, whenever our hash table is filled by 60%, the elements are moved to a new hash table of double the size of the original hash table.

Q. what is the default capacity and load factor of LinkedHashSet in java Collection?

It's possible to create a linked hash set without defining its capacity and load factor. For example,

```
// LinkedHashSet with default capacity and load factor
LinkedHashSet<Integer> numbers1 = new LinkedHashSet<>();
```

By default,

- the capacity of the linked hash set will be 16
- the load factor will be 0.75

How to Create the LinkedHashSet by using other collection

If we want to create the LinkedHashSet by using other collection framework we have to create the object of collection class and store the element in it and we have to create the object of LinkedHashSet and pass the collection reference in its constructor.

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

public class Main {
   public static void main(String[] args) {
      // Creating an arrayList of even numbers
      ArrayList<Integer> evenNumbers = new ArrayList<>();
      evenNumbers.add(2);
      evenNumbers.add(4);
      System.out.println("ArrayList: " + evenNumbers);
      // Creating a LinkedHashSet from an ArrayList
```

```
        LinkedHashSet<Integer> numbers = new
LinkedHashSet<>(evenNumbers);
        System.out.println("LinkedHashSet: " + numbers);
    }
}
```

Output:

ArrayList: [2, 4]
LinkedHashSet: [2, 4]

Q. what is the diff between HashSet and LinkedHashSet in Java?

Internal Working	HashSet internally uses HashMap for storing objects	LinkedHashSet uses LinkedHashMap internally to store objects
When To Use	If you don't want to maintain insertion order but want to store unique objects	If you want to maintain the insertion order of elements then you can use LinkedHashSet
Order	HashSet does not maintain insertion order	LinkedHashSet maintains the insertion order of objects
Complexity of Operations	HashSet gives O(1) complexity for insertion, removing, and retrieving objects	LinkedHashSet gives insertion, removing, and retrieving operations performance in order O(1).
Performance	The performance of HashSet is better when compared to LinkedHashSet and TreeSet.	The performance of LinkedHashSet is slower than TreeSet. It is almost similar to HashSet but slower because LinkedHashSet internally maintains LinkedList to maintain the insertion order of elements

Compare	HashSet uses equals() and hashCode() methods to compare the objects	LinkedHashSet uses equals() and hashCode() methods to compare it's objects
Null Elements	HashSet allows only one null value.	LinkedHashSet allows only one null value.

Q. what are the similarities between HashSet and LinkedHashSet?

- Duplicates: HashSet, LinkedHashSet and TreeSet are implements Set interface, so they are not allowed to store duplicates objects.

- Thread-safe: If we want to use HashSet, LinkedHashSet, and TreeSet in a multi-threading environment then first we make it externally synchronized because both LinkedHashSet and TreeSet are not thread-safe.

- All three are Cloneable and Serializable

Q. **When to use HashSet, TreeSet, and LinkedHashSet in Java:**

1. **HashSet:** If you don't want to maintain insertion order but wants to store unique objects.
2. **LinkedHashSet:** If you want to maintain the insertion order of elements then you can use LinkedHashSet.
3. **TreeSet:** If you want to sort the elements according to some Comparator then use TreeSet.

Q. Explain the All Set Operations With Examples?

Union of Sets: This operation adds all the elements in one set with the other.
Two perform the union between two sets, we can use the addAll() method. Now consider we have the two sets and we want to perform the union operation on it.
Let set1 = [1, 3, 2, 4, 8, 9, 0] and set2 = [1, 3, 7, 5, 4, 0, 7, 5]
After union we have the following result

Union = [0, 1, 2, 3, 4, 5, 7, 8, 9]

Example

```
import java.util.*;
 public class SetExample {
    public static void main(String args[])
   {
      Set<Integer> a = new HashSet<Integer>();
      a.addAll(Arrays.asList( new Integer[] { 1, 3, 2, 4, 8, 9, 0 }));

     // Again declaring object of Set class
     // with reference to HashSet
      Set<Integer> b = new HashSet<Integer>();
     b.addAll (Arrays.asList(new Integer[] { 1, 3, 7, 5, 4, 0, 7, 5 }));
      // To find union
      Set<Integer> union = new HashSet<Integer>(a);
      union.addAll(b);
      System.out.print("Union of the two Set");
      System.out.println(union);
   }
}
```

Intersection of Sets

This operation returns all the common elements from the given two sets.
For the above two sets, the intersection would be
Let set1 = [1, 3, 2, 4, 8, 9, 0] and set2 = [1, 3, 7, 5, 4, 0, 7, 5]
After Intersection: [0, 1, 3, 4]

Example

```
import java.util.*;
 public class SetExample {
    public static void main(String args[])
   {  Set<Integer> a = new HashSet<Integer>();
      a.addAll(Arrays.asList( new Integer[] { 1, 3, 2, 4, 8, 9, 0 }));
```

```
      // Again declaring object of Set class
      // with reference to HashSet
        Set<Integer> b = new HashSet<Integer>();
      b.addAll(Arrays.asList(new Integer[] { 1, 3, 7, 5, 4, 0, 7, 5 }));
      // To find intersection
        Set<Integer> intersection = new HashSet<Integer>(a);
        intersection.retainAll(b);
        System.out.print("Intersection of the two Set");
        System.out.println(intersection);
    }
}
```

Difference: This operation removes all the values present in one set from the other set.

To calculate the difference between the two sets, we can use the removeAll() method.

Let set1 = [1, 3, 2, 4, 8, 9, 0] and set2 = [1, 3, 7, 5, 4, 0, 7, 5]

After Difference: [2, 8, 9]

Example

```
import java.util.*;
public class SetExample {
    public static void main(String args[])
   {  Set<Integer> a = new HashSet<Integer>();
       a.addAll(Arrays.asList( new Integer[] { 1, 3, 2, 4, 8, 9, 0 }));
      // Again declaring object of Set class
      // with reference to HashSet
        Set<Integer> b = new HashSet<Integer>();
      b.addAll(Arrays.asList(new Integer[] { 1, 3, 7, 5, 4, 0, 7, 5 }));
  // To find the symmetric difference
   Set<Integer> difference = new HashSet<Integer>(a);
   difference.removeAll(b);
   System.out.print("Difference of the two Set");
   System.out.println(difference);
    }
}
```

Q. what is the difference between LinkedHashSet VS HashSet?

LinkedHashSet Vs. TreeSet

Here are the major differences between LinkedHashSet and TreeSet:

- The TreeSet class implements the SortedSet interface. That's why elements in a tree set are sorted. However, the LinkedHashSet class only maintains the insertion order of its elements.
- A TreeSet is usually slower than a LinkedHashSet. It is because whenever an element is added to a TreeSet, it has to perform the sorting operation.
- LinkedHashSet allows the insertion of null values. However, we cannot insert a null value to TreeSet

Q. What is TreeSet?

TreeSet is like HashSet which contains the unique elements only but in a sorted manner. The major difference is that TreeSet provides a total ordering of the elements. The elements are ordered using their natural ordering, or by a Comparator typically provided at sorted set creation time. The set's iterator will traverse the set in ascending element order.

Q. Why and when we use TreeSet?

We prefer TreeSet in order to maintain the unique elements in the sorted order .

Q. What is natural ordering in TreeSet?

"Natural" ordering is the ordering implied by the implementation of Comparable interface by the objects in the TreeSet . Essentially RBTree must be able to tell which object is smaller than other object , and there are two ways to supply that logic to the RB Tree implementation :

1. We need to implement the Comparable interface in the class(es) used as objects in TreeSet.
2. Supply an implementation of the Comparator would do comparing outside the class itself.

Q. Why do we need TreeSet when we already had Sorted Set?

SortedSet is an interface while TreeSet is the class implementing it. As we know, in java, we can not create the objects of the interface. The class implementing the interface must fulfill the contract of interface, i.e , concrete class must implement all the methods present in the interface. TreeSet is such an implementation.

Q. What happens if the TreeSet is concurrently modified while iterating the elements?

The iterator's returned by the TreeSet class iterator method are fail-fast. fail-fast means if the set is modified at any time after the iterator is created , in any way except the iterator's own remove method , the iterator will throw a ConcurrentModificationException. Thus , in the face of concurrent modification , the iterator fails quickly and cleanly

Q. Which copy technique (deep or shallow) is used by TreeSet clone() method ?

According to Oracle docs , clone() method returns the shallow copy of the TreeSet instance. In shallow copy , Both objects A and B shared the same memory location .

Q. How to convert HashSet to TreeSet object?

```
Set    treeObject = new TreeSet (hashSetObject);
```