Day11: Collection framework, List, Set Queue, equals and hashCode method, Comparable and Comparator

Collection Interface:

The Collection interface is the interface that is implemented by all the classes in the collection framework. It declares the core/common methods that every collection will have.

They are as follows:

Method	Description
public boolean add(E e)	It is used to insert an element in this collection.
public int size()	It returns the total number of elements in the collection.
public boolean contains(Object element)	It is used to search an element.
public boolean isEmpty()	It checks if collection is empty.
public void clear()	It removes the total number of elements from the collection.
public boolean remove(Object element)	It is used to delete an element from the collection.
public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
public Object[] toArray()	It converts collection into array.
public Iterator iterator()	It returns an Iterator object.
public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
default Stream<E> stream()	It returns a sequential Stream with the collection as its source.
default Stream<E> parallelStream()	It returns a possibly parallel Stream with the collection as its source.
default boolean removeIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.

List Interface:

List interface is the child interface of the Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

- List is index-based, it able to arrange all the elements as per indexing.
- List is able to allow duplicate elements.
- List is following insertion order.

- List is able to allow any number of null values.

In addition to all the methods of the Collection Interface, List interface defines some other methods as well, which are applicable to only List interface implemented classes.

They are :

- **public void add(int index, Object obj)**: It able to add the specified element at the specified index value.
- **public Object set(int index, Object obj):** Replaces the element at the specified position in this list with the specified element.
- **public Object get(int index)**: Returns the element at the specified position in this list.
- **public Object remove(int index):** It will remove and return an element available at the specified index value.
- **public int indexOf(Object obj):** It will return an index value where the first occurrence of the specified element.

Implementation classes of List interface:

1. ArrayList:

The ArrayList class implements the List interface. It uses a dynamic array to store the elements.

It dynamically increase and decrease in size.

ArrayList is the best choice if our frequent operation is retrieval.

Example:

```
ArrayList<String> al = new ArrayList<>();
```

In the above statement we have created an empty ArrayList class object which can hold multiple String objects.

The default initial capacity of the ArrayList is 10.

Once the ArrayList reaches to the maximum capacity, then internally a new ArrayList object will be created in the memory automatically with following formula:

```
newCapacity = (currentCapacity * 3/2)+1;
```

We can create an ArrayList object with different initial capacity as well.

Example:

```
ArrayList<String> al = new ArrayList<>(100); // here initial capacity will be 100
```

Example:

```
import java.util.ArrayList;

class Main {

    public static void main(String args[]) {

        ArrayList<String> al = new ArrayList<>();

        al.add("one");
        al.add("two");
        al.add("three");
        al.add("one");//duplicate element
        al.add(null);
        al.add("four");
        al.add(null); //duplicate null

        System.out.println(al);
        System.out.println(al.size());

    }
}

Output:
[one, two, three, one, null, four, null]
7

Note: All the collection classes internally overrides the toString() method, so when
      we print the object of the collection classes, it will print the elements inside
      the [] square bracket, instead of printing the address.
```

Traversing the List type of collection elements one by one:

List type of collection follows the zero based index, there are many way to traverse the element from the List implemented classes:

Example

1. Using normal for-loop:

```
for(int i=0;i<al.size();i++){
      System.out.println(al.get(i));
   }
```

2. Using enhanced for-loop:

```
for(String s: al){
      System.out.println(s);
  }
```

3. Using Iterator:

```
      Iterator<String> itr= al.iterator();

        while(itr.hasNext()){
            String s= itr.next();
            System.out.println(s);
        }
```

   etc.

Note: All the collection classes does not allows the primitives types, it only accept the objects.

so if we want to store any primitive values, we need to store it in the form of corresponding wrapper classes. Internally collection classes uses auto-boxing and auto-unboxing feature to store the primitive in the form of corresponding wrapper classes objects.

Example:

```
ArrayList<Integer> al = new ArrayList<>();

al.add(10);
al.add(12);
```

Auto boxing and Auto unboxing:

The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing. This is the new feature of Java5. So java programmer doesn't need to write the conversion code.

The main advantage of Autoboxing and unboxing is, no need of conversion between primitives and Wrappers manually so less coding is required.

Example

```
int x = 10;
//lets convert this primitive to the corresponding wrapper object

Integer i1 = Integer.valueOf(x); // boxing

Integer i2 = x; //autoboxing

or

Integer i3 = 10; //autoboxing

Unboxing:

int x = i3.intValue(); //unboxing

int x = i3; // auto-unboxing
```

Example:

```
import java.util.ArrayList;

class Main {

    public static void main(String args[]) {

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

        al.add(10);
        al.add(20);
        al.add(30);
```

```
        al.add(40);
        al.add(50);


        for(int x: al){
            System.out.println(x);
        }
    }
}
```

Note: we can call all the methods defined inside the Collection interface on the ArrayList object.

Example: Searching an element:

```
import java.util.ArrayList;

class Main {

    public static void main(String args[]) {

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

        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);

        System.out.println(al.contains(20));
        System.out.println(al.contains(80));
    }
}

Output:
true
false
```

Example: converting an ArrayList object to the Object array

```
import java.util.ArrayList;

class Main {

    public static void main(String args[]) {

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

        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);

          Object[] or= al.toArray();

          for(Object o:or){
              int x= (Integer)o;
              System.out.println(x);
          }
    }
}
```

I Problem:

Let’s create an application, where user will be prompted to enter the Student details

(roll, name, marks) till the user opted out. once he will opted out, then display all the student details back.

```
//Student bean class
//Student.java
public class Student {

    private int roll;
    private String name;
    private int marks;

    public Student() {
    }

    public Student(int roll, String name, int marks) {
        this.roll = roll;
        this.name = name;
        this.marks = marks;
    }

    public int getRoll() {
        return roll;
    }

    public void setRoll(int roll) {
        this.roll = roll;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getMarks() {
        return marks;
    }

    public void setMarks(int marks) {
        this.marks = marks;
    }
}


//Main.java

import java.util.ArrayList;
import java.util.Scanner;

class Main {

    public static void main(String args[]) {

        ArrayList<Student> students = new ArrayList<>();
        Scanner sc = new Scanner(System.in);

        int count = 1;

        while (true){
            System.out.println("Enter the details of Student "+(count++));

            System.out.println("Enter Roll");
            int roll= sc.nextInt();

            System.out.println("Enter Name :");
            String name= sc.next();

            System.out.println("Enter marks");
            int marks= sc.nextInt();
```

```
            Student student = new Student(roll,name,marks);

            students.add(student);

            System.out.println("Want More (y/n) ?");
            String choice = sc.next();

            if(choice.equalsIgnoreCase("n"))
                break;
        }


        System.out.println("Printing Details of all Students");
        System.out.println("================================");
        for(Student student: students){

            System.out.println("Roll is :"+student.getRoll());
            System.out.println("Name is :"+student.getName());
            System.out.println("Marks is :"+student.getMarks());

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

        }
    }
}
```

LinkedList class:

This class is another implementation of the List interface, it internally uses the **doubly linked list data structure.** we can add and remove data from both end.

This class is almost the same as ArrayList class, i.e. it also maintains the insertion order, and allows the duplicate element.

LinkedList class is the best choice if our frequent operation is insertion or deletion of the elements from the middle because no shifting is required.

fig- doubly linked list

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

Here, The capacity concept is not applicable.

Vector class:

This class is also one of the implementation classes of List interface.

This class is also same as the ArrayList class with following differences:

Methods of the ArrayList class is not synchronized, whereas most of the methods of the Vector class is synchronized.

Example:

```
Vector<String> v = new Vector<>();
```

Stack class:

The stack is the subclass of the Vector class. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of the Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Example:

```
import java.util.Stack;

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

        Stack<String> stack = new Stack<String>();

        stack.push("A");
        stack.push("B");
        stack.push("C");
        stack.push("D");
        stack.push("E");

        stack.pop(); //remove the last element

        for(String s:stack){
            System.out.println(s);
        }
    }
}

Output:
A
B
C
D
```

Note: To the variable of List we can store any of its implementation object by this way we can achieve the Runtime polymorphism.

Example:

```
List<String> list1 = new ArrayList<>();
List<String> list2 = new LinkedList<>();
List<String> list3 = new Vector<>();
List<String> list4 = new Stack<>();
```

Set Interface:

This interface extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items.

This Set is implemented by **HashSet, LinkedHashSet,** and **TreeSet** classes.

```
Set<data-type> s1 = new HashSet<>();
Set<data-type> s2 = new LinkedHashSet<>();
Set<data-type> s3 = new TreeSet<>();
```

HashSet class:

HashSet class implements Set Interface.

It does not allow the duplicate elements and Insertion order will not be preserved.

At most one null value we can add in HashSet object.

If our frequent operation is searching, then HashSet class is the best choice, because it internally uses hashing technique to store the objects.

O(1):- is the time complexity of searching any element by using hashing.

Example:

```
HashSet<String> hs = new HashSet<>();
```

Example:

```
import java.util.HashSet;

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

        HashSet<String> hs = new HashSet<>();

        hs.add("Delhi");
        hs.add("Mumbai");
        hs.add("Chennai");
        hs.add("Pune");
        hs.add("Delhi"); //duplicate element
        hs.add(null); //adding null
        hs.add("Bangaluru");


        System.out.println(hs);
    }
}

Output:
[null, Delhi, Chennai, Bangaluru, Pune, Mumbai]
```

Note: Since Set does not follow the index, so we can not access the elements one by one from any of the Set implemented classes, so to traverse the elements of the Set implemented classes

we can not use normal for-loop, but we can use enhanced for-loop or Iterator to traverse elements of a Set implemented classes.

LinkedHashSet class:

It is the child class of the HashSet class .

This class is similar to the HashSet class, but it will preserve the insertion order.

Example:

```
import java.util.LinkedHashSet;

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

        LinkedHashSet<String> lhs = new LinkedHashSet<>();

        lhs.add("Delhi");
        lhs.add("Mumbai");
        lhs.add("Chennai");
        lhs.add("Pune");
        lhs.add("Delhi"); //duplicate element
        lhs.add(null); //adding null
        lhs.add("Bangaluru");


        System.out.println(lhs);
    }
}

Output:
[Delhi, Mumbai, Chennai, Pune, null, Bangaluru]
```

equals() and hashcode() method in Java:

The *Object* class defines both the *equals()* and *hashCode()* methods, which means that these two methods are implicitly defined in every Java class, including the ones we create:

We need to override these two methods inside our java classes if we want to make 2 object of our java class logically equal.

Example

```
Student s1 = new Student(10,"Ram",500); //roll,name,marks
Student s2 = new Student(10,"Ram",500); //roll,name,marks

Note: technically at memory level both objects are different, but if we want to make
```

```
both object s1 and s2 equal logically then we need to override both
method inside the Student class.
```

equals() method:

- The java equals() is a method of *lang.Object* class, and it is used to compare two objects.
- To compare two objects that whether they are the same, it compares the values of both the object's attributes.
- By default, two objects will be the same only if stored in the same memory location.

Syntax:

```
public boolean equals(Object obj)
```

It takes the reference object as the parameter, with which we need to make the comparison. and returns the true if both the objects are the same, else returns false.

General Contract of equals() method:

There are some general principles defined by Java.

that must be followed while implementing the equals() method in Java. The equals() method must be:

- ***reflexive***: An object x must be equal to itself, which means, for object x, **equals(x)** should return true.
- ***symmetric***: for two given objects x and y, *x.equals(y)* must return true if and only if *y.equals(x)* returns true.
- ***transitive***: for any objects x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- ***consistent***: for any objects x and y, the value of x.equals(y) should change, only if the property in equals() changes.
- For any object x, the *equals(null)* must return false.

  In Object class this equals() method is defiened as follows:

```
public boolean equals(Object o){
      if(o == null)
        return false;
      else
          return this == o;

}
```

hashCode() method:

- A **hashcode** is an integer value associated with every object in Java, facilitating the hashing in hash tables.

- To get this hashcode value for an object, we can use the hashcode() method in Java. It is the means ***hashcode() method that returns the integer hashcode value of the given object***.
- Since this method is defined in the Object class, hence it is inherited by user-defined classes also.
- The hashcode() method returns the same hash value when called on two objects, which are equal according to the equals() method. And if the objects are unequal, it usually returns different hash values.

Syntax:

```
public int hashCode()
```

It returns the hash code value for the given objects.

Contract for hashcode() method in Java:

If two objects are the same as per the equals(Object) method, then if we call the hashCode() method on each of the two objects, it must provide the same integer result.

Note: As per the Java documentation, both the methods should be overridden to get the complete equality mechanism; using equals() alone is not sufficient. It means, if we override the equals(), we must override the hashcode() method.

Note: **HashSet** and **LinkedHashSet** class internally equals and hashCode method to identify the duplicate objects.

We Problem:

Let's Override the equals and hashCode method inside the Student class, to make 2 student object equal logically if their roll, name and marks is same. so that if we try to add 2 student objects inside a HashSet or LinkedHashSet class, then it will ignore the duplicate student object.

```
//Student.java
public class Student {

    private int roll;
    private String name;
    private int marks;

    public Student() {
    }

    public Student(int roll, String name, int marks) {
        this.roll = roll;
        this.name = name;
        this.marks = marks;
    }


    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
```

```
        Student s1 = this;
        Student s2 = (Student) o;

        return s1.roll == s2.roll && s1.marks == s2.marks && s1.name.equals(s2.name);

    }

    @Override
    public int hashCode() {
        return roll;
    }

    public int getRoll() {
        return roll;
    }

    public void setRoll(int roll) {
        this.roll = roll;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getMarks() {
        return marks;
    }

    public void setMarks(int marks) {
        this.marks = marks;
    }

@Override
    public String toString() {
        return "Student{" +
                "roll=" + roll +
                ", name='" + name + '\'' +
                ", marks=" + marks +
                '}';
    }

}
```

```
//Main.java

import java.util.HashSet;
import java.util.Set;

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

        Set<Student> students = new HashSet<>();

        students.add(new Student(10,"Ram",500));
        students.add(new Student(12,"Vishnu",600));
        students.add(new Student(10,"Ram",500)); //duplicate object
        students.add(new Student(14,"Srinu",600));

        System.out.println(students.size());

        System.out.println(students);
    }
}

Output:
3
[Student{roll=10, name='Ram', marks=500}, Student{roll=12, name='Vishnu', marks=600}, Student{roll=14, name='Srinu', marks=600}]
```

TreeSet class:

This class is the implementation of Set and SortedSet interface.

- In this implementation, objects are sorted and stored in ascending order according to their natural order.
- The *TreeSet* uses a self-balancing binary search tree.
- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class doesn't allow null element. even a single null is also not allowed otherwise it throws NullPointerException at runtime.

Example:

```
import java.util.TreeSet;
class Main{
    public static void main(String args[]){

        TreeSet<String> ts=new TreeSet<String>();

        ts.add("Delhi");
        ts.add("Mumbai");
        ts.add("Chennai");
        ts.add("Bangaluru");

        System.out.println(ts);
    }
}

Output:
[Bangaluru, Chennai, Delhi, Mumbai] //In sorted order
```

Note: TreeSet class by default only accept the **Comparable** object. if we try to add any non-comparable object inside the TreeSet object, then at runtime it will throw a **ClassCastException.**

Comparable interface in java:

In Java, Comparable is an interface belongs to **java.lang** package. which is having only one method.

```
public int compareTo(Object obj);
```

In order to add our User-defined java classes object inside the TreeSet, we should implement the Comparable interface inside our User-defined class and need to specify the sorting logic by overriding this compareTo method.

The compareTo method is used to compare the current object with the specified object. It returns:

- positive integer, if the current object is greater than the specified object.
- negative integer, if the current object is less than the specified object.

- zero, if the current object is equal to the specified object.

Note: All the Wrapper classes and String class internally implements the Comparable interface.

Example: implementing the Comparable interface inside the Student class. According to the roll number.

```
package com.masai;


public class Student implements Comparable<Student>{

    private int roll;
    private String name;
    private int marks;

    public Student() {
    }

    public Student(int roll, String name, int marks) {
        this.roll = roll;
        this.name = name;
        this.marks = marks;
    }



    public int getRoll() {
        return roll;
    }

    public void setRoll(int roll) {
        this.roll = roll;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getMarks() {
        return marks;
    }

    public void setMarks(int marks) {
        this.marks = marks;
    }

    @Override
    public String toString() {
        return "Student{" +
                "roll=" + roll +
                ", name='" + name + '\'' +
                ", marks=" + marks +
                '}';
    }

    @Override
    public int compareTo(Student student) {

        if(this.roll > student.roll)
            return +1;
        else if(this.roll < student.roll)
            return -1;
        else
```

```
            return 0;

    }
}
```

Now we can add the Student object inside the TreeSet object.

Example:

```
import java.util.TreeSet;

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

        TreeSet<Student> ts = new TreeSet<>();

        ts.add(new Student(20,"Amit",520));
        ts.add(new Student(15,"Suresh",550));
        ts.add(new Student(22,"Ajay",540));
        ts.add(new Student(18,"Rajesh",590));

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

Overriding to compareTo method to sort the object according to the name.

```
@Override
    public int compareTo(Student student) {

       return name.compareTo(student.name);

    }
```

Comparator interface in java:

If we want to define the sorting logic for objects of some class, outside that class then we can use Comparator interface.

This Comparator interface belongs to **java.util** package.this interface also has only abstract method :

```
public int compare(Object obj1, Object obj2);
```

Let’s define the sorting logic of Student object outside the Student class, i.e. inside the another class using Comparator interface.

```
//StudentRollComparator.java
import java.util.Comparator;

public class StudentRollComparator implements Comparator<Student> {
```

```
    @Override
    public int compare(Student s1, Student s2) {

        if(s1.getRoll() > s1.getRoll())
            return +1;
        else if(s1.getRoll() < s2.getRoll())
            return -1;
        else
            return 0;

    }
}
```

Now we can add the Student object inside the TreeSet collection object by mentioning this “StudentRollComparator” class in the constructor of the TreeSet object.

In this case Student object need not implement the Comparable interface any more.

Example:

```
import java.util.TreeSet;

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

        TreeSet<Student> ts = new TreeSet<>(new StudentRollComparator());

        ts.add(new Student(20,"Amit",520));
        ts.add(new Student(15,"Suresh",550));
        ts.add(new Student(22,"Ajay",540));
        ts.add(new Student(18,"Rajesh",590));

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

Difference between Comparable and Comparator:

Comparable	**Comparator**
Comparable interface belongs to java.lang package	Comparator interface belongs to java.util package
If we want to specify the sorting logic of a class object within the same class , we need to use Comparable	If we want to specify the sorting logic of a class object outside that class then we should use Comparator.
With the help of Comparable we can define only one sorting logic within a class.	With the help of Comparator we can define multiple sorting logic of a class object inside multiple classes.
Here method name is: public int compareTo(Object obj)	Here method name is : public int compare(Object obj1, Object obj2)

You Problem:

- Create a Employee Bean class with following 3 fields:

  int empId, String name, int salary.

- Implement the Comparable interface within the Employee class and define the sorting logic according to the salary, and if 2 Employee salary is same then sort them according to their name.

- Create a Main class with main method and inside the main class create a TreeSet class object which can hold multiple Employee object.
- Add 5 Employee object with some dummy date and print each Each Employee object one by one.