```
Interfaces
++++++++++
1. Collection

2. List   -> ArrayList,LinkedList   :: add(int, object), set(int,object),
remove(int,object)
        -> LegacyClasses :: Stack,Vector
        -> CopyOnWriteArrayList[Avoids ConcurrentModification]

3. Set    -> HashSet,LinkedHashSet  :: add(object),set(object),remove(object)
          -> CopyOnWriteArraySet[Avoids ConcurrentModification]

4. NavigableSet -> TreeSet
5. SortedSet    -> TreeSet
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++
6. Map(K,V) ->
HashMap,LinkedHashMap,WeakHashMap,IdentityHashMap,Hastable,Properties ::
put(object,object),get(Object)
            -> ConcurrentHashMap[Avoids ConcurrentModification]


7. NavigableMap -> TreeMap
8. SortedMap    -> TreeMap

Interfaces related to Sorting
+++++++++++++++++++++++++++
1. Comparable  :: meant for DefaultNatural Sorting order of Objects :: int
compareTo(Object obj)
2. Comparator  :: meant for Custom Sorting :: int compare(Object obj1,Object obj2)

Cursors related to iteration
++++++++++++++++++++++++++
1. Enumeration  :: iterators for legacy classes[read operation only]
2. Iterator       :: Universal cursor[read in forward direction and removal
operation]  :: hasNext(),next()
3. ListIterator :: iterators for List
objects[read(forward,backward),read,update,remove]

UtilityClasses
++++++++++++++
1.Collections   :: sort(),binarySearch(),reverse(),reverseOrder()
2.Arrays    :: sort(),binarySearch(),asList()



LambdaExpression in Collection
==============================
 1. List(I)
      -> ArrayList,LinkedList,Vector,Stack
      -> Insertion order is preserved.
      -> Duplicate Objects are allowed.

 2. Set(I)
      -> HashSet,TreeSet(based on some sorting)
      -> Insertion order is not preserved becoz data would be stored based on
HashCode.
      -> Duplicate Objects are not allowed.
```

3. Map(I)
-> HashMap,TreeMap(based on some sorting)
-> Data would be stored in the form of K,V pair
-> Insertion order is not preserved becoz data would be stored based on HashCode of Key.
-> Duplicate Objects are not allowed.

Comparator(I)
-> It contains only one abstract method called "compare()".
-> To define our own sorting we need to use Customized Sorting.
-> public int compare(Object obj1,Object obj2)
=> return -ve iff obj1 has to come before obj2.
=> return +ve iff obj1 has to come after obj2.
=> return 0 iff obj1 and obj2 are equal.

SortedCollections
==========
1. Sorted List
2. Sorted Map
3. Sorted Set

Sorted List
==========
List(may be ArrayList,LinkedList,Vector or Stack) never talks about sorting order.
If we want sorting for the list then we should use Collections class sort() method.
Collections.sort(list)==>meant for Default Natural Sorting Order
For String objects: Alphabetical Order
For Numbers : Ascending order

Demo Program to Sort elements of ArrayList according to Defaut Natural Sorting Order
(Ascending Order):

```
1) import java.util.ArrayList;
2) import java.util.Collections;
3) class Test
4) {
5)    public static void main(String[] args)
6)    {
7)          ArrayList<Integer> l = new ArrayList<Integer>();
8)                l.add(10);
9)                l.add(0);
10)               l.add(15);
11)               l.add(5);
12)               l.add(20);
13)         System.out.println("Before Sorting:"+l);
14)         Collections.sort(l);
15)         System.out.println("After Sorting:"+l);
16)   }
17) }
```

Output:
Before Sorting:[10, 0, 15, 5, 20]
After Sorting:[0, 5, 10, 15, 20]

Instead of Default natural sorting order if we want customized sorting order then we should go for Comparator interface.
Comparator interface contains only one abstract method: compare()
Hence it is Functional interface.
public int compare(obj1,obj2)

returns -ve iff obj1 has to come before obj2
returns +ve iff obj1 has to come after obj2
returns 0 iff obj1 and obj2 are equal

Collections.sort(list,Comparator)==>meant for Customized Sorting Order

Demo Program to Sort elements of ArrayList according to Customized Sorting Order (Descending Order):

```
1) import java.util.TreeSet;
2) import java.util.Comparator;
3) class MyComparator implements Comparator<Integer>
4) {
5)    public int compare(Integer I1,Integer I2)
6)    {
7)          if(I1<I2)
8)          {
9)                return +1;
10)         }
11)         else if(I1>I2)
12)         {
13)               return -1;
14)         }
15)         else
16)         {
17)               return 0;
18)         }
19)   }
20) }
21) class Test
22) {
23)   public static void main(String[] args)
24)   {
25)         TreeSet<Integer> l = new TreeSet<Integer>(new MyComparator());
26)                l.add(10);
27)                l.add(0);
28)                l.add(15);
29)                l.add(5);
30)                l.add(20);
31)               System.out.println(l);
32)    }
33) }
```

//Descending order Comparator
Output: [20, 15, 10, 5, 0]

Shortcut way:

```
1) import java.util.ArrayList;
2) import java.util.Comparator;
3) import java.util.Collections;
4) class MyComparator implements Comparator<Integer>
5) {
6)    public int compare(Integer I1,Integer I2)
7)    {
8)          return (I1>I2)?-1:(I1<I2)?1:0;
9)    }
10) }
11) class Test
12) {
```

```
13)   public static void main(String[] args)
14)   {
15)         ArrayList<Integer> l = new ArrayList<Integer>();
16)               l.add(10);
17)               l.add(0);
18)               l.add(15);
19)               l.add(5);
20)               l.add(20);
21)         System.out.println("Before Sorting:"+l);
22)         Collections.sort(l,new MyComparator());
23)         System.out.println("After Sorting:"+l);
24)   }
25) }
```

Sorting with Lambda Expressions
==========

As Comparator is Functional interface, we can replace its implementation with Lambda Expression

```
Collections.sort(l,(I1,I2)->(I1<I2)?1:(I1>I2)?-1:0);
```

Demo Program to Sort elements of ArrayList according to Customized Sorting Order By using
Lambda Expressions(Descending Order):

```
1) import java.util.ArrayList;
2) import java.util.Collections;
3) class Test
4) {
5)    public static void main(String[] args)
6)      {
7)          ArrayList<Integer> l= new ArrayList<Integer>();
8)               l.add(10);
9)               l.add(0);
10)              l.add(15);
11)              l.add(5);
12)              l.add(20);
13)         System.out.println("Before Sorting:"+l);
14)         Collections.sort(l,(I1,I2)->(I1<I2)?1:(I1>I2)?-1:0);
15)         System.out.println("After Sorting:"+l);
16)      }
17) }
```

Output:

```
Before Sorting:[10, 0, 15, 5, 20]
After  Sorting:[20, 15, 10, 5, 0]
```

Shortcut code
++++++++++++

```
import java.util.*;

public class Test
{
   public static void main(String[] args)
      {
            ArrayList<Integer> al = new ArrayList<Integer>();
            al.add(10);
            al.add(0);
            al.add(5);
            al.add(15);
```

```
            al.add(20);

            System.out.println("Before Sorting :: "+al);

            //Functional Programming :: passing function body as argument
            Collections.sort(al,(i1,i2)->-i1.compareTo(i2));

            System.out.println("After Sorting  :: "+al);
    }
}
```

```
Output
Before Sorting :: [10, 0, 5, 15, 20]
After Sorting  :: [20, 15, 10, 5, 0]
```

```
Sorting for Customized class objects by using Lambda Expressions
================================================================
```

```
without lambda Expression
=========================
```

```
import java.util.*;

class Employee
{
      int eno;
      String ename;

      Employee(int eno,String ename){
            this.eno=eno;
            this.ename=ename;
      }
      public String toString(){
            return eno+"==>"+ename;
      }
}
class EmployeeComparator implements Comparator
{
      public int compare(Object obj1,Object obj2){
            Employee e1=(Employee) obj1;
            Employee e2=(Employee) obj2;

            return e1.eno>e2.eno? -1 : e1.eno<e2.eno ? +1:0;
      }
}
public class Test {
      public static void main(String[] args){
            ArrayList<Employee> al=new ArrayList<Employee>();
            al.add(new Employee(10,"sachin"));
            al.add(new Employee(14,"ponting"));
            al.add(new Employee(7,"dhoni"));
            al.add(new Employee(9,"lara"));
            al.add(new Employee(18,"kohli"));
            System.out.println("Before sorting :"+al);

            Collections.sort(al,new EmployeeComparator());

            System.out.println("After sorting :"+al);
```

```
   }
}
Before sorting :[10==>sachin, 14==>ponting, 7==>dhoni, 9==>lara, 18==>kohli]
After sorting :[18==>kohli, 14==>ponting, 10==>sachin, 9==>lara, 7==>dhoni]


With Lambda Expression
======================
import java.util.*;
class Employee
{
      int eno;
      String ename;

      Employee(int eno,String ename){
            this.eno=eno;
            this.ename=ename;
      }
      public String toString(){
            return eno+"==>"+ename;
      }
}
public class Test {
      public static void main(String[] args){

            ArrayList<Employee> al=new ArrayList<Employee>();
            al.add(new Employee(10,"sachin"));
            al.add(new Employee(14,"ponting"));
            al.add(new Employee(7,"dhoni"));
            al.add(new Employee(9,"lara"));
            al.add(new Employee(18,"kohli"));
            System.out.println("Before sorting :"+al);

            Collections.sort(al,(e1,e2)-> e1.eno>e2.eno?-1:e1.eno<e2.eno?+1:0);

            System.out.println("After sorting :"+al);

   }
}
Before sorting :[10==>sachin, 14==>ponting, 7==>dhoni, 9==>lara, 18==>kohli]
After sorting :[18==>kohli, 14==>ponting, 10==>sachin, 9==>lara, 7==>dhoni]


Shortcut code
++++++++++++
class Employee
{
      Integer eno;
      String ename;

      Employee(Integer eno,String ename)
      {
            this.eno = eno;
            this.ename =ename;
      }

      public String toString()
      {
            return eno + "---" + ename;
```

```
      }
}

//Client Code
public class Test
{
   public static void main(String[] args)
      {
            ArrayList<Employee> al = new ArrayList<Employee>();
                  al.add(new Employee(10,"sachin"));
                  al.add(new Employee(7,"dhoni"));
                  al.add(new Employee(18,"kohli"));
                  al.add(new Employee(19,"dravid"));
                  al.add(new Employee(45,"rohith"));

            System.out.println("Before Sorting :: "+al);

            //Functional Programming :: passing function body as argument
            Collections.
                  sort(al,
                        //(e1,e2)-> (e1.eno > e2.eno) ? -1 : (e1.eno < e2.eno) ?
+1 :0
                        (e1,e2) -> -e1.eno.compareTo(e2.eno)
                       );

            System.out.println("After Sorting  :: "+al);
    }
}
output
Before Sorting :: [10---sachin, 7---dhoni, 18---kohli, 19---dravid, 45---rohith]
After Sorting  :: [45---rohith, 19---dravid, 18---kohli, 10---sachin, 7---dhoni]

2. Sorted Set

In the case of Set, if we want Sorting order then we should go for TreeSet
1. TreeSet t = new TreeSet();
      This TreeSet object meant for default natural sorting order
2. TreeSet t = new TreeSet(Comparator c);
      This TreeSet object meant for Customized Sorting Order

Demo Program for Default Natural Sorting Order(Ascending Order):
1) import java.util.TreeSet;
2) class Test
3) {
4)    public static void main(String[] args)
5)    {
6)          TreeSet<Integer> t = new TreeSet<Integer>();
7)                t.add(10);
8)                t.add(0);
9)                t.add(15);
10)               t.add(5);
11)               t.add(20);
12)         System.out.println(t);
13)   }
14) }
Output: [0, 5, 10, 15, 20]

Demo Program for Customized Sorting Order(Descending Order):
1) import java.util.TreeSet;
```

```
2) class Test
3) {
4)    public static void main(String[] args)
5)    {
6)          TreeSet<Integer> t = new TreeSet<Integer>((I1,I2)->(I1>I2)?-1:(I1<I2)?
1:0);
7)                t.add(10);
8)                t.add(0);
9)                t.add(15);
10)               t.add(25);
11)               t.add(5);
12)               t.add(20);
13)         System.out.println(t);
14)   }
15) }
Output: [25, 20, 15, 10, 5, 0]
```

3. Sorted Map:

In the case of Map, if we want default natural sorting order of keys then we should go for TreeMap.

1. TreeMap m = new TreeMap();
   This TreeMap object meant for default natural sorting order of keys
2. TreeMap t = new TreeMap(Comparator c);
   This TreeMap object meant for Customized Sorting Order of keys

Demo Program for Default Natural Sorting Order(Ascending Order):

```
import java.util.*;

public class Test {
      public static void main(String[] args){
            TreeMap<Integer,String> t =new TreeMap<Integer,String>();
            t.put(10,"sachin");
            t.put(14,"ponting");
            t.put(18,"kohli");
            t.put(9,"lara");
            t.put(17,"ABD");
            t.put(7,"dhoni");
            System.out.println(t);
   }
}
{7=dhoni, 9=lara, 10=sachin, 14=ponting, 17=ABD, 18=kohli}
```

Demo Program for Customized Sorting Order(Descending Order):

```
import java.util.*;

public class Test {
      public static void main(String[] args){
        TreeMap<Integer,String> t =new TreeMap<Integer,String>((I1,I2)->I1>I2?-
1:I1<I2?+1:0);
            t.put(10,"sachin");
            t.put(14,"ponting");
            t.put(18,"kohli");
            t.put(9,"lara");
            t.put(17,"ABD");
            t.put(7,"dhoni");

            System.out.println("After sorting ::"+t);
   }
```

}
After sorting ::{18=kohli, 17=ABD, 14=ponting, 10=sachin, 9=lara, 7=dhoni}

Stream API
++++++++++
 package name : java.util.stream.*;

Streams
+++++++
To process objects of the collection, in 1.8 version Streams concept introduced.

What is the differences between java.util.streams and java.io streams?
=> java.util streams meant for processing objects from the collection. Ie, it represents a stream of objects from the collection.

=> Java.io streams meant for processing binary and character data with respect to file. i.e it represents stream of
   binary data or character data from the file.

=> Hence java.io streams and java.util streams both are different.

What is the difference between collection and stream?
 => If we want to represent a group of individual objects as a single entity then We should go for collection.

 => If we want to process a group of objects from the collection then we should go for streams.

 => We can create a stream object to the collection by using stream() method of Collection interface.
    stream() method is a default method added to the Collection in 1.8 version.

=> Stream is an interface present in java.util.stream. Once we got the stream, by using that we can process objects of that collection.
      We can process the objects in the following 2 phases
      1.Configuration
      2.Processing

1) Configuration:
      We can configure either by using filter mechanism or by using map mechanism.

Filtering:
      We can configure a filter to filter elements from the collection based on some boolean condition by using
      filter()method of Stream interface.
                  public Stream filter(Predicate<T> t)
                  here (Predicate<T > t) can be a boolean valued function/lambda expression
                           |-> boolean test(T t)

Mapping:
      If we want to create a separate new object, for every object present in the collection based on our requirement then we should go for
      map() method of Stream interface.

```
                  public Stream map (Function f); // It can be lambda expression
also
                                 |-> R apply(T t)
Ex:
 Stream s = c.stream();
 Stream s1 = s.map(i-> i+10);
 Once we performed configuration we can process objects by using several methods.


eg#1.
import java.util.*;
import java.util.stream.*;

//Client Code
public class Test
{
   public static void main(String[] args)
      {
         ArrayList<Integer> al = new ArrayList<Integer>();
                  al.add(0);
                  al.add(5);
                  al.add(10);
                  al.add(15);
                  al.add(20);
                  al.add(25);

                  ArrayList<Integer> doubleList = new ArrayList<Integer>();
                  for ( Integer i1: al )
                        doubleList.add(i1*2);

                  System.out.println(doubleList);

                  System.out.println("****Using Stream API*****");
                  List<Integer> stremCode =
                        al.stream() //creating a stream
                        .map(i->i*2)//Configuration : new object
                        .collect(Collectors.toList());//Creating a list
                  System.out.println(streamCode);
      }
}

Output
[0, 10, 20, 30, 40, 50]
****Using Stream API*****
[0, 10, 20, 30, 40, 50]


eg#1.
Using Stream API
+++++++++++++++
ArrayList<Integer> al = new ArrayList<Integer>();
                  al.add(0);
                  al.add(5);
                  al.add(10);
                  al.add(15);
                  al.add(20);
                  al.add(25);

System.out.println("******Using Stream API******");
```

```
      List<Integer> streamList =
                  al.stream()   //created a stream
                    .filter(i->i%2==0)//filtering the data
                    .collect(Collectors.toList());//collecting as List
       System.out.println(streamList);
```

Output

```
******Using Stream API******
 [0,10,20]
```

eg#2.

```
import java.util.*;
import java.util.stream.*;

//Client Code
public class Test
{
   public static void main(String[] args)
      {
         List<Integer> al = new ArrayList<Integer>();

            al.add(10);
            al.add(15);
            al.add(20);
            al.add(0);
            al.add(100);
            al.add(25);

            int result=sumIterator(al);
            System.out.println("The sum is :: "+result);

            System.out.println("Using Stream API");
            System.out.println("The sum is :: "+sumIteratorUsingStream(al));

      }

      public static int sumIterator(List<Integer> al)
      {
            //Without using Stream API
            int sum = 0;
            Iterator<Integer> itr = al.iterator();
            while (itr.hasNext())
            {
                  int num  = itr.next();
                  if (num>10)
                  {
                        sum+=num;
                  }
            }
            return sum;
      }
      public static int sumIteratorUsingStream(List<Integer> al)
      {
            //Using Stream API
            return al.stream().filter(i->i>10).mapToInt(i->i).sum();
      }
}
```

eg#2.

```
import java.util.*;
import java.util.stream.*;

//Client Code
public class Test
{
   public static void main(String[] args)
      {
         List<String> al = new ArrayList<String>();
         al.add("sachin");
         al.add("saurav");
         al.add("dhoni");
         al.add("kohli");
         al.add("yuvi");

         printObjectData(al);

         printObjectDataUsingStream(al);


      }

      public static void printObjectData(List<String> al)
      {
            List<String> result = new ArrayList<String>();
            //Without using Stream API
            for (String name :al )
            {
                  result.add(name.toUpperCase());
            }
            System.out.println(result);

      }

      public static void printObjectDataUsingStream(List<String> al)
      {
            //Using Stream API
            List<String> result =
                  al.stream()
                    .map(name->name.toUpperCase())
                    .collect(Collectors.toList());
            System.out.println(result);
      }
}
```

Output

```
[SACHIN, SAURAV, DHONI, KOHLI, YUVI]
[SACHIN, SAURAV, DHONI, KOHLI, YUVI]
```

2) Processing

processing by collect() method

```
import java.util.*;
import java.util.stream.*;

class Employee
```

```
{
      int eid;
      String ename;
      float esal;

      Employee(int eid,String ename,float esal)
      {
            this.eid   = eid;
            this.ename = ename;
            this.esal  = esal;
      }

      public String toString()
      {
            return eid+"->"+ename+"->" +esal;
      }

}

//Client Code
public class Test
{
   public static void main(String[] args)
      {
         List<Employee> al = new ArrayList<Employee>();
         al.add(new Employee(10,"sachin",35000f));
         al.add(new Employee(7,"dhoni",30000f));
         al.add(new Employee(18,"kohli",35000f));
         al.add(new Employee(1,"rahul",28000f));
         al.add(new Employee(19,"dravid",30000f));
         al.add(new Employee(45,"rohith",25000f));

         System.out.println(al);

         printObjectDataUsingStream(al);
      }

      public static void printObjectDataUsingStream(List<Employee> al)
      {
            //Using Stream API
            List<Float> result =
                  al.stream()
                  .filter(emp->emp.esal<35000f)
                  .map(emp->emp.esal)
                  .collect(Collectors.toList());
            System.out.println(result);

            System.out.println();

            Set<Float> noDuplicates =
                  al.stream()
                    .filter(emp->emp.esal<35000f)
                    .map(emp->emp.esal)
                    .collect(Collectors.toSet());

            System.out.println(noDuplicates);

            System.out.println();
```

```
            Map<Integer,String> map =
            al.stream()
              .filter(emp->emp.esal<35000f)
            .collect(Collectors.toMap(emp->emp.eid,emp->emp.ename));
            System.out.println(map);

      }
}
```

```
Output
[10->sachin->35000.0, 7->dhoni->30000.0, 18->kohli->35000.0, 1->rahul->28000.0, 19-
>dravid->30000.0, 45->rohith->25000.0]
[30000.0, 28000.0, 30000.0, 25000.0]
[25000.0, 30000.0, 28000.0]
{1=rahul, 19=dravid, 7=dhoni, 45=rohith}
```

b.Processing by count()method

count()

======

This method returns number of elements present in the stream.

```
      public long count()
```

c.Processing by sorted()method

If we sort the elements present inside stream then we should go for sorted() method.

The sorting can either default natural sorting order or customized sorting order specified by comparator.

```
            sorted()- default natural sorting order
            sorted(Comparator c)-customized sorting order.
```

d.Processing by min() and max() methods

```
   min(Comparator c)
      returns minimum value according to specified comparator.
   max(Comparator c)
      returns maximum value according to specified comparator.
```

e.forEach() method

This method will not return anything.

This method will take lambda expression as argument and apply that lambda expression for each element present in the stream.

f.toArray() method

We can use toArray() method to copy elements present in the stream into specified array

g.Stream.of()method

We can also apply a stream for group of values and for arrays.

eg#1.

```
import java.util.*;
import java.util.stream.*;
```

```
class Employee
{
	int eid;
	String ename;
	float esal;

	Employee(int eid,String ename,float esal)
	{
		this.eid   = eid;
		this.ename = ename;
		this.esal  = esal;
	}

	public String toString()
	{
		return "["+eid+","+ename+","+esal+"]";
	}

}

//Client Code
public class Test
{
   public static void main(String[] args)
	{
	   List<Employee> al = new ArrayList<Employee>();
	   al.add(new Employee(10,"sachin",35000f));
	   al.add(new Employee(7,"dhoni",30000f));
	   al.add(new Employee(18,"kohli",35000f));
	   al.add(new Employee(1,"rahul",28000f));
	   al.add(new Employee(19,"dravid",30000f));
	   al.add(new Employee(45,"rohith",25000f));

	   System.out.println(al);

	   printObjectDataUsingStream(al);
	}

	public static void printObjectDataUsingStream(List<Employee> al)
	{
		System.out.println();

		//Using Stream API
		long result = al.stream().filter(emp->emp.esal>30000).count();
		System.out.println(result);

		System.out.println();
		//forEach(Consumer consumer) :: void accept(T t)
		al.stream().forEach(emp->System.out.println(emp));

		System.out.println();
		//MethodReference :: instance
		al.stream().forEach(System.out::println);

		System.out.println();
		ArrayList<Integer> al1 =new ArrayList<Integer>();
			al1.add(0);
			al1.add(10);
			al1.add(5);
```

```
                al1.add(20);
                al1.add(15);
                System.out.println(al1);

            Integer[] arr = al1.stream().toArray(Integer[] :: new);
                for (int data:arr)
                {
                      System.out.println(data);
                }
      }
}
```

Output

```
[[10,sachin,35000.0], [7,dhoni,30000.0], [18,kohli,35000.0], [1,rahul,28000.0],
[19,dravid,30000.0], [45,rohith,25000.0]]

2

[10,sachin,35000.0]
[7,dhoni,30000.0]
[18,kohli,35000.0]
[1,rahul,28000.0]
[19,dravid,30000.0]
[45,rohith,25000.0]

[10,sachin,35000.0]
[7,dhoni,30000.0]
[18,kohli,35000.0]
[1,rahul,28000.0]
[19,dravid,30000.0]
[45,rohith,25000.0]
```

eg#2.

```
import java.util.*;
import java.util.stream.*;

class Employee implements Comparable
{
      Integer eid;
      String ename;
      float esal;

      Employee(Integer eid,String ename,float esal)
      {
            this.eid   = eid;
            this.ename = ename;
            this.esal  = esal;
      }

      public String toString()
      {
            return "["+eid+","+ename+","+esal+"]";
      }

      @Override
      public int compareTo(Object obj1)
      {
            Employee e2 = (Employee) obj1;
```

```
            int eid1 = this.eid;
            int eid2 = e2.eid;

            return eid1<eid2 ? -1 : eid1>eid2 ? +1 :0 ;
      }

}

public class Test
{
   public static void main(String[] args)
      {
         List<Employee> al = new ArrayList<Employee>();

         al.add(new Employee(10,"sachin",35000f));
         al.add(new Employee(7,"dhoni",30000f));
         al.add(new Employee(18,"kohli",35000f));
         al.add(new Employee(1,"rahul",28000f));
         al.add(new Employee(19,"dravid",30000f));
         al.add(new Employee(45,"rohith",25000f));

         System.out.println(al);

         printObjectDataUsingStream(al);
      }

      public static void printObjectDataUsingStream(List<Employee> al)
      {
            System.out.println();

            al.stream()
              .sorted()
              .collect(Collectors.toList())
              .forEach(System.out::println);

            System.out.println();

         System.out.println("Sorting in Descending Order");
            al.stream()
              .sorted((e1,e2)->-e1.eid.compareTo(e2.eid))
              .collect(Collectors.toList())
              .forEach(System.out::println);

          System.out.println();
            Employee emp1=al.stream()
                  .min((e1,e2)->e1.esal.compareTo(e2.esal))
                  .get();
            System.out.println("Min salary employee :: "+emp1);

            System.out.println();
            Employee emp2=al.stream()
                  .max((e1,e2)->e1.esal.compareTo(e2.esal))
                  .get();
            System.out.println("Max salary employee :: "+emp2);
      }
}
```

Output

```
[[10,sachin,35000.0], [7,dhoni,30000.0], [18,kohli,35000.0], [1,rahul,28000.0],
[19,dravid,30000.0], [45,rohith,25000.0]]

2

[10,sachin,35000.0]
[7,dhoni,30000.0]
[18,kohli,35000.0]
[1,rahul,28000.0]
[19,dravid,30000.0]
[45,rohith,25000.0]

[10,sachin,35000.0]
[7,dhoni,30000.0]
[18,kohli,35000.0]
[1,rahul,28000.0]
[19,dravid,30000.0]
[45,rohith,25000.0]
```