Collection framework in JAVA

Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.

A Collection represents a single unit of objects, i.e., a group.

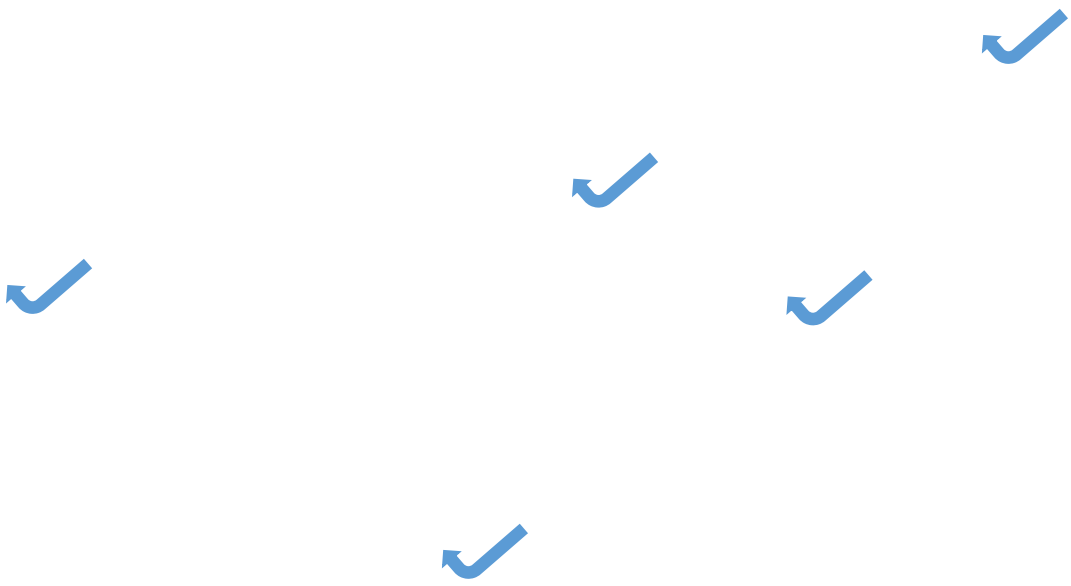
What is a framework in Java

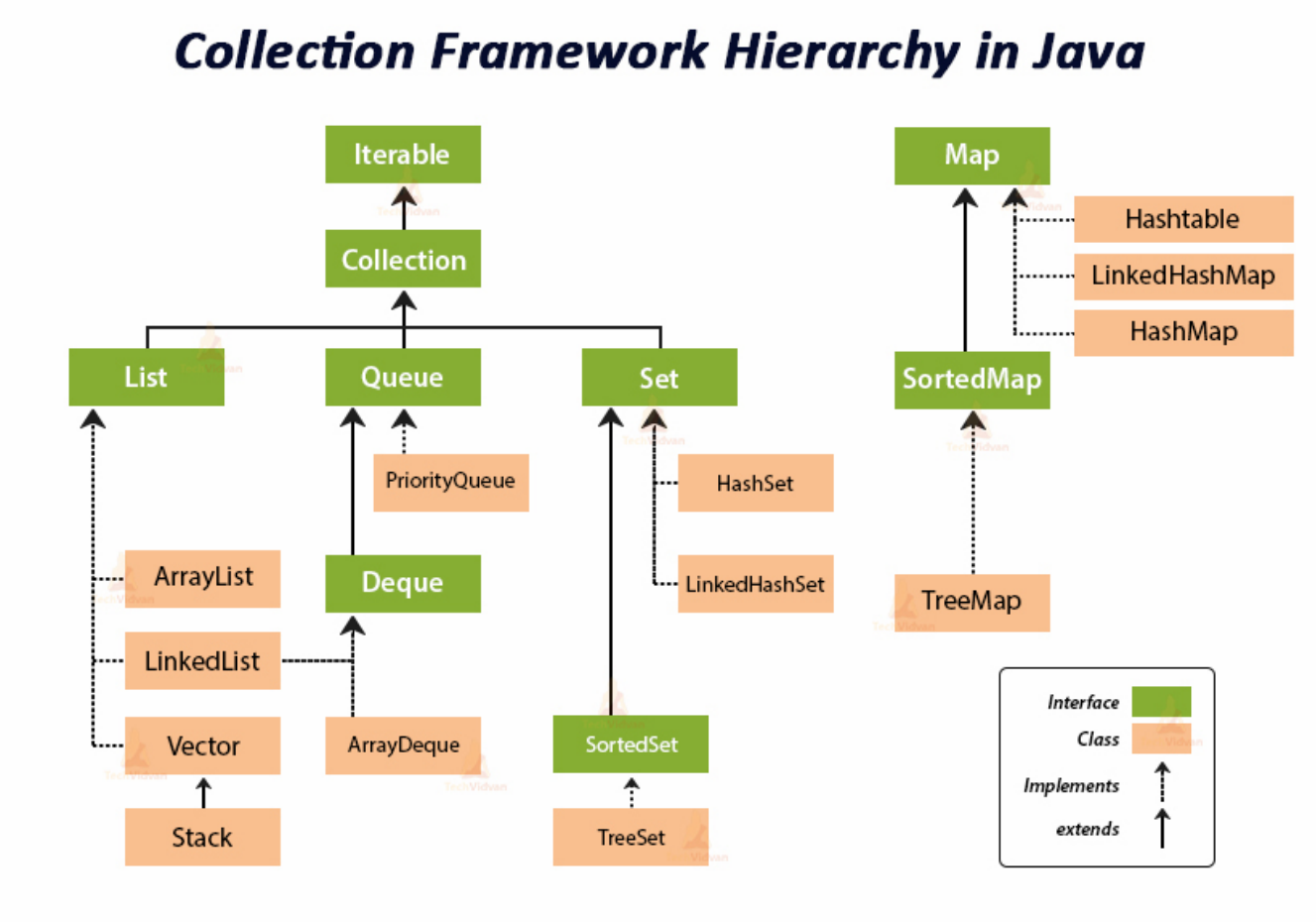
- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm





1. Explain Map interface in detail.
2. Which classes implement Set interface? Explain any one in detail. (HashSet , TreeSet)
3. Explain the following collection classes.
    a. ArrayList
    b. HashMap
    c. HashSet
    d. TreeSet
    e. TreeMap
4. Difference between Hashset and Treeset.
5. Difference between Comparable and comparator interface in JAVA.

Collection interface

No.	Method	Description
1	public boolean **add**(E e)	It is used to insert an element in this collection.
2	public boolean **addAll**(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean **remove(**Object element)	It is used to delete an element from the collection.
4	public boolean **removeAll**(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	public int **size**()	It returns the total number of elements in the collection.
6	public void **clear**()	It removes the total number of elements from the collection.
7	public **Iterator** iterator()	It returns an iterator.
8	public boolean **isEmpty()**	It checks if collection is empty.
9	default Stream<E> stream()	It returns a sequential Stream with the collection as its source.
10	public boolean equals(Object element)	It matches two collections.
11	public int hashCode()	It returns the hash code number of the collection

Iterator interface

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.

ArrayList :- ArrayList class extends abstract class AbstractList and implements the List interface. ArrayList is a generic class that has this declaration:

class ArrayList<E>

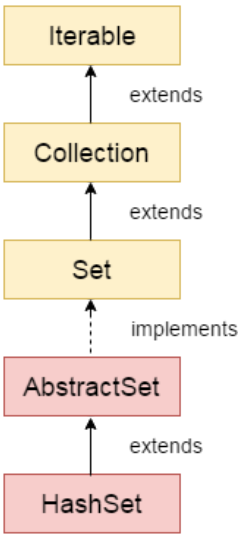
- Growable Array implementation of List interface.
- Insertion order is preserved.
- Duplicate elements are allowed.
- Multiple null elements of insertion are allowed.
- Default initial capacity of an ArrayList is 10.
- No Thread safe: Multiple threads can access the array list at the same time. There are no synchronized methods.

When to use ArrayList: Use ArrayList if you want to retrieve the elements frequently. Because ArrayList implements RandomAccess interface and its index based collection.

```
import java.util.*;

public class DemoArrayList {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		ArrayList<Integer> al=new ArrayList<Integer>();
		al.add(1);
		al.add(2);
		al.add(5);
		System.out.println("Size of al = "+al.size());
		System.out.println("Contents of al = " +al);
		System.out.println("=====After sorting=====");
		Collections.sort(al);
		for(int i=0;i<al.size();i++)
			System.out.println(al.get(i));

		ArrayList<Integer> bl=new ArrayList<Integer>();
		bl.addAll(al);

		System.out.println("Both collections are equals? = " + al.equals(bl));

		bl.remove(1);
		System.out.println("Size of bl after remove a number = "+bl.size());
		Iterator<Integer> bi=bl.iterator();
		while(bi.hasNext())
			System.out.println(bi.next());
	}
}
```

Note: add() , remove() ,addAll() , removeAll() , clear() , size(),contains() ,

HashSet:



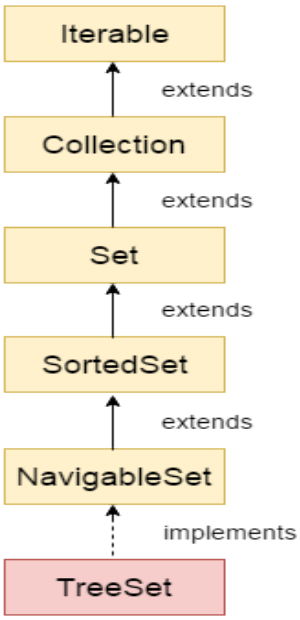
HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

- HashSet stores the elements by using a mechanism called hashing
- Insertion order is not preserved.
- Duplicates are not allowed.
- Allows null element
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16, and the load factor is 0.75.
- Heterogeneous objects allowed.

```
import java.util.*;

public class DemoHashset {

	public static void main(String args[]){
		HashSet<String> set=new HashSet<String>();
	    set.add("Ravi");
	    set.add("Vijay");
	    set.add("Arun");
	    set.add("Sumit");
	    set.add("Vijay");
	    System.out.println("An initial list of elements: "+set);

	    //Removing specific element from HashSet
	    set.remove("Ravi");
	    System.out.println("After invoking remove(object) method: "+set);

	    //check If an Item Exists
	    System.out.println(set.contains("Vijay"));

	    HashSet<String> set1=new HashSet<String>();
	    set1.add("Ajay");
	    set1.add("Gaurav");
	    set.addAll(set1);
	    System.out.println("Updated List: "+set);

	    //Removing all the new elements from HashSet
	    set.removeAll(set1);
	    System.out.println("After invoking removeAll() method: "+set);

	    //Removing all the elements available in the set
	    set.clear();
	    System.out.println("After invoking clear() method: "+set);
	 }
}
```

Note: add() , remove() ,addAll() , removeAll() , clear() , size()

TreeSet :



TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

- Underlying data structure is balanced tree.
- Duplicates are not allowed.
- Insertion order is not preserved since objects will be inserted based on some sorting technique.
- Null insertion is not possible in a non-empty TreeSet. We will get NullPointerException if we add.
- Heterogeneous objects are not allowed. We will get ClassCastException if we add.
- **Homogenous and Comparable:** If you need default sorted order, the **objects** which you are adding in a TreeSet should be Homogenous and Comparable. Otherwise we will get ClassCaseException.

```
import java.util.*;

public class DemoTreeSet {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Set<String> ts=new TreeSet<String>();
		ts.add("12");
		ts.add("dfs");
		System.out.println(ts);
		ts.remove("12");
		System.out.println(ts);
	}
}
```

Difference Between HashSet and TreeSet

Parameters	HashSet	TreeSet
Ordering or Sorting	It does not provide a guarantee to sort the data.	It provides a guarantee to sort the data. The sorting depends on the supplied Comparator.
Null Objects	In HashSet, **only an element** can be null.	It does not allow null elements.
Comparison	It uses **hashCode()** or **equals()** method for comparison.	It uses **compare()** or **compareTo()** method for comparison.
Performance	It is **faster** than TreeSet.	It is **slower** in comparison to HashSet.
Implementation	Internally it uses **HashMap** to store its elements.	Internally it uses **TreeMap** to store its elements.
Data Structure	HashSet is backed up by a hash table.	TreeSet is backed up by a Red-black Tree.
Values Stored	It allows **heterogeneous** value.	It allows only **homogeneous** value.

There are some similarities between HashSet and TreeSet:

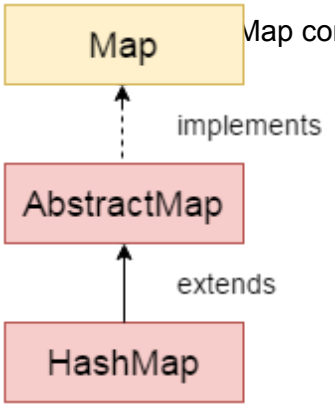
- Both the classes implement the Set interface.
- They do not allow duplicate values.
- Both HashSet and TreeSet are not thread-safe.
- They are not synchronized but if we want to make them synchronize, we can use Collections.synchronizedSet() method.

HashMap :-

HashMap implements Map<K, V>, Cloneable and Serializable interface. It extends AbstractMap<K, V> class. It belongs to java.util package.

K: It is the type of keys maintained by this map.

V: It is the type of mapped values.



Map contains values based on the key.

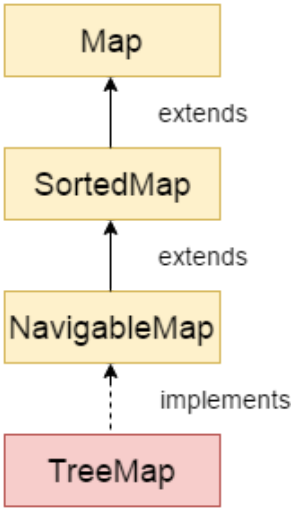
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap is non synchronized.
- Java HashMap maintains no order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

```
import java.util.*;

public class DemoHashMap {
    public static void main(String[] args) {
            // TODO Auto-generated method stub
            Map<Integer, String> hp=new HashMap<Integer, String>();
            hp.put(1,"Gandhinagar");
            hp.put(5, "Goa");

            for(Integer k :hp.keySet())
                  System.out.println("Key = " + k + " Value = " + hp.get(k));
            hp.put(5, "Goaa"); //Update recent entry with duplicate key
            hp.putIfAbsent(2, "Anand");
            System.out.println("After add an element if not in map :");
            for(Integer k :hp.keySet())
                  System.out.println("Key = " + k + " Value = " + hp.get(k));

            //create one more Hashmap and copy all element from existing
            Map<Integer, String> hp1=new HashMap<Integer, String>();
            hp1.putAll(hp);
            System.out.println("After invoking putAll() method with hp1");
            for(Integer k :hp.keySet())
                  System.out.println("Key = " + k + " Value = " + hp.get(k));

            //Remove element
            hp1.remove(2);
            System.out.println("After invoking remove() method with hp1");
            for(Integer k :hp.keySet())
                  System.out.println("Key = " + k + " Value = " + hp.get(k));

            //Replace element
            hp1.replace(1, "Ahmedabad");
```

```
            hp1.replace(5, "Goaa", "Goa");
            System.out.println("After invoking replace() method with hp1");
            for(Integer k :hp.keySet())
                  System.out.println("Key = " + k + " Value = " + hp.get(k));
      }
   }
```

Note : put() , putIfAbsent(), putAll(), replace(),

TreeMap:- TreeMap class extends **AbstractMap<K, V>** class and implements **NavigableMap<K, V >**, **Cloneable**, and **Serializable** interface. TreeMap is an example of a **SortedMap**. It is implemented by the Red-Black tree, which means that the order of the keys is sorted.



- TreeMap also contains value based on the key.
- TreeMap is sorted by keys.
- It contains unique elements.
- It cannot have a null key but have multiple null values.
- Keys are in ascending order.
- It stores the object in the tree structure.

```
import java.util.*;

public class DemoTreeMap {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		TreeMap<Integer, String> tm=new TreeMap<Integer, String>();
		tm.put(101, "Anand");
		tm.put(104, "Goa");

		for(Map.Entry m: tm.entrySet())
			System.out.println(m.getKey() + " " + m.getValue());

		tm.put(103, "Kheda");
		System.out.println(tm);

		//Returns key-value pairs whose keys are less than the specified key.
		System.out.println("headMap: "+tm.headMap(102));
		//Returns key-value pairs whose keys are greater than or equal to the specified
key.
		System.out.println("tailMap: "+tm.tailMap(102));
		//Returns key-value pairs exists in between the specified key.
		System.out.println("subMap: "+tm.subMap(100, 102));
	}
}
```

Note : put() , putall() , headMap() , tailMap() , subMap() , firstKey() , lastKey()

What is difference between HashMap and TreeMap?

Basis	HashMap	TreeMap
Definition	Java **HashMap** is a hashtable based implementation of Map interface.	Java **TreeMap** is a Tree structure-based implementation of Map interface.
Interface Implements	HashMap implements **Map, Cloneable**, and **Serializable** interface.	TreeMap implements **NavigableMap, Cloneable**, and **Serializable** interface.
Null Keys/ Values	HashMap allows a **single** null key and **multiple** null values.	TreeMap does not allow **null** keys but can have **multiple** null values.
Homogeneous/ Heterogeneous	HashMap allows heterogeneous elements because it does not perform sorting on keys.	TreeMap allows homogeneous values as a key because of sorting.
Performance	HashMap is **faster** than TreeMap	TreeMap is **slow** in comparison to HashMap
Data Structure	The HashMap class uses the **hash table**.	TreeMap internally uses a **Red-Black** tree, which is a self-balancing Binary Search Tree.
Functionality	HashMap class contains only basic functions like **get(), put(), KeySet()**, etc. .	TreeMap class is rich in functionality, because it contains functions like: **tailMap(), firstKey(), lastKey(), pollFirstEntry(), pollLastEntry()**.
Order of elements	HashMap does not maintain any order.	The elements are sorted in **natural order** (ascending).
Uses	The HashMap should be used when we do not require key-value pair in sorted order.	The TreeMap should be used when we require key-value pair in sorted (ascending) order.

Similarities between HashMap and TreeMap

- **HashMap** and **TreeMap** classes implement **Cloneable** and **Serializable** interface.
- Both the classes extend **AbstractMap<K, V>** class.
- A Map is an object which stores **key-value** pairs. In the key-value pair, each key is unique, but their values may be **duplicate**.
- Both classes represents the mapping from **key** to **values**.
- Both maps are not **synchronized**.

- Map use **put()** method to add an element in the map.
- The iterator throws a **ConcurrentModificationException** if the map gets modify in any way.

Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

Comparable Program :

```
public class Democomparable  {
	public static void main(String[] args) {
		TreeSet<Student> t=new TreeSet<Student>();
		t.add(new Student("Ram","Patel"));
		t.add(new Student("Anand","Patel"));
		t.add(new Student("Bijal","Pandya"));
		t.add(new Student("Aazad","Shah"));
		t.add(new Student("Anand","Sharma"));
		System.out.println(t);
	}
}

class Student implements Comparable<Student>
{
	String name;
	String lname;
	Student(String name,String lname)
	{
		this.name=name;
		this.lname=lname;
	}

	public int compareTo(Student o) {
		if(this.name.compareTo(o.name)!=0)
			return this.name.compareTo(o.name);
		else
			return this.lname.compareTo(o.lname);
	}

	@Override
```

```
	public String toString() {
		return  name +  " " + lname;
	}
}
```

Program 2:

```
import java.util.*;

public class DemoComparableList {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		List<student1> slist=new ArrayList<student1>();
		slist.add(new student1(5,300));
		slist.add(new student1(3,750));
		slist.add(new student1(1,450));
		System.out.println("========Before sorting By total ==========");
		for(student1 s:slist)
			System.out.println(s);

		Collections.sort(slist);
		System.out.println("========After sorting By total ==========");
		for(student1 s:slist)
			System.out.println(s);
	}
}

class student1 implements Comparable<student1>
{
	int id,total;
	student1(int id, int total)
	{
		this.id=id;
		this.total=total;
	}
	@Override
	public String toString() {
		// TODO Auto-generated method stub
		return "Student id=" + id +  ", total=" + total ;
	}
	@Override
	public int compareTo(student1 o) {
		if (total>o.total)
			return 1;
		else
			return -1;
	}
}
```

Comparator Program :

Program 1:

```
import java.util.*;

public class DemoCollection {

	public static void main(String args[]){
		TreeSet set=new TreeSet(new MyOrder());
	  set.add(new String("Ravi"));
	  set.add(new String("Vijay"));
	  set.add(new String("Arun"));
	  set.add(new String("Sumit"));

	  System.out.println("An initial list of elements: "+set);
	}
}
class MyOrder implements Comparator<String>
{
	@Override
	public int compare(String o1, String o2)
	{
		String a=(String) o1;
		String b=(String) o2;
		return b.compareTo(a);
	}
}
```

Program 2(a):

```
package Comparator;

import java.util.*;

public class DemoComaratorList {


		public static void main(String[] args) {
			// TODO Auto-generated method stub
			List<student> slist=new ArrayList<student>();
			slist.add(new student(5,"Ram",300));
			slist.add(new student(3,"Shyam",550));
			slist.add(new student(1,"Neeta",450));
			System.out.println("========Before sorting By total ==========");
			for(student s:slist)
				System.out.println(s);
			sortByTotal st=new sortByTotal();
			Collections.sort(slist, st);
			System.out.println("========After sorting By total ==========");
			for(student s:slist)
				System.out.println(s);
			sortByName st1=new sortByName();
			Collections.sort(slist, st1);
			System.out.println("========After sorting By total ==========");
			for(student s:slist)
```

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

        class student
        {
                int id,total;
                String name;
                student(int id, String name,int total)
                {
                        this.id=id;
                        this.name=name;
                        this.total=total;
                }
                @Override
                public String toString() {
                        // TODO Auto-generated method stub
                        return "Student id=" + id + " Name = " + name + ", total=" + total ;
                }

        }
        class sortByTotal implements Comparator<student>
        {
                @Override
                public int compare(student o1, student o2) {
                        if(o1.total>o2.total)
                                return 1;
                        else
                                return -1;
                }
        }
        class sortByName implements Comparator<student>
        {
                @Override
                public int compare(student o1, student o2) {
                        return o1.name.compareTo(o2.name);
                }
        }
```

Program 2(b):

```
import java.util.*;

public class DemoComparator {

        public static void main(String[] args) {
                TreeSet<Student1> t=new TreeSet<Student1> (new SortByLname());
                t.add(new Student1("Ram","Patel"));
                t.add(new Student1("Anand","Patel"));
                t.add(new Student1("Bijal","Pandya"));
```

```
		t.add(new Student1("Aazad","Shah"));
		t.add(new Student1("Anand","Sharma"));
		System.out.println(t);
	}
}

class SortByLname implements Comparator<Student1>
{
	@Override
	public int compare(Student1 o1, Student1 o2) {
		if(o1.name.compareTo(o2.name)!=0)
			return o1.name.compareTo(o2.name);
		else
			return o1.lname.compareTo(o2.lname);
	}
}

class Student1 {
	String name;
	String lname;
	Student1(String name,String lname)
	{
		this.name=name;
		this.lname=lname;
	}

	@Override
	public String toString() {
		return name +  " " + lname;
	}
}
```