1. Shyam has to submit an assignment on SET interface, the assignment is to perform union, intersection and difference operation on SET, help Shyam complete the assignment.

**Aim:** The aim of this Java program is to help Shyam complete his assignment related to the **Set** interface. The program demonstrates how to perform the following set operations on two sets: union, intersection, and difference.

**Algorithm:**

1. Create two **HashSet** objects, **a** and **b**, and populate them with integer values.
2. To find the union of the two sets, create a new **HashSet** called **union** and add all elements from both sets to it.
3. To find the intersection of the two sets, create a new **HashSet** called **intersection** and retain only the elements that exist in both sets using the **retainAll()** method.
4. To find the difference between the two sets (elements present in set **a** but not in set **b**), create a new **HashSet** called **difference** and remove all elements from set **b** using the **removeAll()** method.
5. Print the results of each operation.

**Program Explanation:**

1. In this program, two **HashSet** objects, **a** and **b**, are created and populated with integer values. **a** contains the elements [1, 2, 3, 4, 5, 6, 10], and **b** contains the elements [1, 3, 7, 5, 4, 0, 7, 5].
2. The program first calculates the union of the two sets by creating a new **HashSet** called **union**, which is initialized with all elements from set **a**. Then, the program adds all elements from set **b** to the **union** set.
3. Next, the program calculates the intersection of the two sets by creating a new **HashSet** called **intersection**. This set is initialized with all elements from set **a**, and then the **retainAll()** method is used to retain only the elements that exist in both set **a** and set **b**.
4. Finally, the program calculates the difference between the two sets by creating a new **HashSet** called **difference**, which is initialized with all elements from set **a**. The **removeAll()** method is then used to remove all elements that are present in set **b** from the **difference** set.
5. The program prints the results of each operation: the union, intersection, and difference of the two sets.

This program provides a clear example of how to perform common set operations using the **Set** interface and demonstrates the concepts of union, intersection, and difference between sets.

Top of Form

**Program:**

**package** labsheet11;

//Java code for demonstrating union, intersection and difference on Set

**import** java.util.\*;

**public** **class** Set\_example{

**public** **static** **void** main(String args[]){

HashSet<Integer> a = **new** HashSet<Integer>();

a.addAll(Arrays.*asList*(**new** Integer[] {1,2,3,4,5,6,10}));

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);

//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);

//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);

}

}

**Output:**

Union of the two Set[0, 1, 2, 3, 4, 5, 6, 7, 10]

Intersection of the two Set[1, 3, 4, 5]

Difference of the two Set[2, 6, 10]

# Develop an application to add elements to a MAP interface and display the key and value pair.

**Aim:** The aim of this Java program is to develop an application that demonstrates how to add key-value pairs to a **Map** interface (specifically a **HashMap**) and then display each key-value pair.

**Algorithm:**

1. Create a **HashMap** object, **hm**, with key-value pairs where keys are strings and values are integers.
2. Add key-value pairs to the **hm** object using the **put(key, value)** method.
3. Retrieve a set view of the key-value pairs using **entrySet()** method.
4. Iterate through the set of key-value pairs using a for-each loop.
5. Inside the loop, access and display each key and its corresponding value.

**Program Explanation:**

1. In this program, a **HashMap** named **hm** is created with key-value pairs. Keys are strings ("a", "b", "c", "d"), and values are integers (150, 200, 250, 300).
2. Key-value pairs are added to the **hm** object using the **put(key, value)** method.
3. The **entrySet()** method is used to retrieve a set view of the key-value pairs, which contains objects of type **Map.Entry<String, Integer>**.
4. The program uses a for-each loop to iterate through the set of key-value pairs. In each iteration, a key-value pair is accessed through the **me** variable, which is of type **Map.Entry<String, Integer>**.
5. Inside the loop, the program extracts the key and value using the **getKey()** and **getValue()** methods of the **Map.Entry** interface and prints them to the console.

This program demonstrates how to work with a **HashMap**, add key-value pairs, and display the key and value for each entry in the map, providing a clear example of the basic usage of a map data structure.

Top of Form

**package** labsheet11;

**package** labsheet11;

**import** java.util.\*;

**class** HashMapDemo{

**public** **static** **void** main(String args[]){

Map< String,Integer> hm =

**new** HashMap< String,Integer>();

hm.put("a", 150);

hm.put("b", 200);

hm.put("c", 250);

hm.put("d", 300);

Set< Map.Entry< String,Integer> > st = hm.entrySet();

/\*

Iterator<Map.Entry<String, Integer>> iterator = st.iterator();

while (iterator.hasNext()) {

Map.Entry<String, Integer> me = iterator.next();

System.out.print(me.getKey() + ":");

System.out.println(me.getValue());

}

\*/

// Returns Set view

**for** (Map.Entry< String,Integer> me:st){

System.***out***.print(me.getKey()+":");

System.***out***.println(me.getValue());

}

}

}

Output:

a:150

b:200

c:250

d:300

1. **Java Comparator example to sort Student object by name, age, and fees. In order to sort Student object on different criteria, we need to create multiple comparators e.g. NameComparator, AgeComparator, and FeesComparator, this is known as custom sorting in Java.**

**/\*This is different from the natural ordering of objects, provided by the compareTo() method of java.lang.Comparable interface. Though both compare() and compareTo() method looks similar they are different in the sense that, former accepts one parameter.\*/**

**Aim:** The aim of this Java program is to demonstrate the use of the **Comparator** interface to sort a collection of **Student** objects based on different criteria, including name, age, and fees. The program creates a list of **Student** objects, sorts them using custom comparator classes, and displays the sorted results.

**Algorithm:**

1. Define a **Student** class with attributes like roll number, name, fees, branch, year, semester, age, and a static variable **clg** representing the college name.
2. Create custom comparator classes (**AgeComparator**, **NameComparator**, and **FeesComparator**) that implement the **Comparator** int erface to specify different sorting criteria for **Student** objects.
3. In the **main** method:
   * Create an **ArrayList** named **sl** to store **Student** objects.
   * Add **Student** objects to the list with various attributes.
   * Sort the list using the custom comparator classes for name, age, and fees criteria.
   * Iterate through the sorted list and display the attributes of each **Student** object.

**Program Explanation:**

1. The **Student** class is defined with attributes representing student information and a static **clg** variable to hold the college name.
2. Three custom comparator classes (**AgeComparator**, **NameComparator**, and **FeesComparator**) implement the **Comparator** interface to define how **Student** objects should be compared for sorting based on age, name, and fees.
3. In the **main** method, an **ArrayList** named **sl** is created to store **Student** objects. Five **Student** objects are added to the list, each with different attributes.
4. The program demonstrates sorting based on different criteria:
   * Sorting by name: It uses the **NameComparator** to sort the list, and the results are displayed.
   * Sorting by age: It uses the **AgeComparator** to sort the list, and the results are displayed.
   * Sorting by fees: It uses the **FeesComparator** to sort the list, and the results are displayed.

Each sorted result is printed, displaying the roll number, name, fees, branch, year, semester, age, and the college name for each **Student** object.

This program showcases the flexibility of custom comparators for sorting objects based on different criteria, allowing you to organize and display data in various ways.

**Program:**

**package labsheet11;**

**import java.util.\*;**

**class Student {**

**int rollno;**

**String name;**

**float fees;**

**String branch;**

**int** year;

**int** sem;

**int** age;

**static** String *clg*;

**public** Student(**int** rollno,String name,**float** fees,String branch,**int** year,**int** sem, **int** age) {

**this**.rollno = rollno;

**this**.name = name;

**this**.fees = fees;

**this**.branch = branch;

**this**.year = year;

**this**.sem = sem;

**this**.age = age;

*clg*="PU";

}

@Override

**public** String toString() {

**return** rollno + " "+ name + " " + fees + " " + branch + " " + year + sem + " " + age + " " + *clg* + "\n";

}

}

**class** AgeComparator **implements** Comparator {

**public** **int** compare(Object o1,Object o2){

Student s1=(Student)o1;

Student s2=(Student)o2;

**if**(s1.age==s2.age)

**return** 0;

**else** **if**(s1.age>s2.age)

**return** 1;

**else**

**return** -1;

}

}

**class** NameComparator **implements** Comparator{

**public** **int** compare(Object o1,Object o2){

Student s1=(Student)o1;

Student s2=(Student)o2;

**return** s1.name.compareTo(s2.name);

}

}

**class** FeesComparator **implements** Comparator {

**public** **int** compare(Object o1,Object o2){

Student s1=(Student)o1;

Student s2=(Student)o2;

**if**(s1.fees==s2.fees)

**return** 0;

**else** **if**(s1.fees>s2.fees)

**return** 1;

**else**

**return** -1;

}

}

**public** **class** PUStudent {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

ArrayList sl=**new** ArrayList();

sl.add(**new** Student(123,"Shiva",10000.00f,"CSE",1,1,18));

sl.add(**new** Student(124,"Venky",15000.00f,"ISE",1,2,20));

sl.add(**new** Student(125,"Jesus",17000.00f,"ECE",1,1,19));

sl.add(**new** Student(126,"Alla",12000.00f,"EEE",1,1,19));

sl.add(**new** Student(127,"Budha",11000.00f,"mech",1,1,21));

System.***out***.println("Sorting by Name");

System.***out***.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

Collections.*sort*(sl,**new** NameComparator());

Iterator itr=sl.iterator();

**while**(itr.hasNext()){

Student st=(Student)itr.next();

System.***out***.println(st.rollno+" "+st.name+" "+ st.fees+ " " + st.branch+ " " + st.year + " " + st.sem + " " + st.age + " " + Student.*clg*);

}

System.***out***.println("Sorting by age");

System.***out***.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

Collections.*sort*(sl,**new** AgeComparator());

Iterator itr2=sl.iterator();

**while**(itr2.hasNext()){

Student st=(Student)itr2.next();

System.***out***.println(st.rollno+" "+st.name+" "+ st.fees+ " " + st.branch+ " " + st.year + " " + st.sem + " " + st.age + " " + Student.*clg*);

}

System.***out***.println("Sorting by fees");

System.***out***.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

Collections.*sort*(sl,**new** FeesComparator());

Iterator itr1=sl.iterator();

**while**(itr1.hasNext()){

Student st=(Student)itr1.next();

System.***out***.println(st.rollno+" "+st.name+" "+ st.fees+ " " + st.branch+ " " + st.year + " " + st.sem + " " + st.age + " " + Student.*clg*);

}

}

}

**Output:**

Sorting by Name

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

126 Alla 12000.0 EEE 1 1 19 PU

127 Budha 11000.0 mech 1 1 21 PU

125 Jesus 17000.0 ECE 1 1 19 PU

123 Shiva 10000.0 CSE 1 1 18 PU

124 Venky 15000.0 ISE 1 2 20 PU

Sorting by age

\_\_\_\_\_\_\_\_\_\_\_\_\_\_

123 Shiva 10000.0 CSE 1 1 18 PU

126 Alla 12000.0 EEE 1 1 19 PU

125 Jesus 17000.0 ECE 1 1 19 PU

124 Venky 15000.0 ISE 1 2 20 PU

127 Budha 11000.0 mech 1 1 21 PU

Sorting by fees

\_\_\_\_\_\_\_\_\_\_\_\_\_\_

123 Shiva 10 000.0 CSE 1 1 18 PU

127 Budha 11000.0 mech 1 1 21 PU

126 Alla 12000.0 EEE 1 1 19 PU

124 Venky 15000.0 ISE 1 2 20 PU

125 Jesus 17000.0 ECE 1 1 19 PU