Experiment 6

Name: Abhishek Shukla
UID: 22BCS16882
Branch: BE-CSE
Section: 22BCS_IOT_639-A
Semester:6th
Date of Performance: 7 March 2025
Subject Name: Project Based Learning in Java with Lab
Subject Code: 22CSH-359

1. **Aim:** Write a program to sort a list of Employee objects (name, age, salary) using lambda expressions.

2. **Algorithm:**

   a) Define an `Employee` class with attributes: `name`, `id`, `age`, and `salary`.

   b) Create a constructor to initialize these attributes.

   c) Implement getter methods for each attribute.

   d) Override the `toString()` method to format the employee details for printing.

   e) In the `main` method, create a list of `Employee` objects with sample data.

   f) Sort employees by salary in descending order using a lambda expression.

   g) Print the sorted list of employees.

   h) Sort employees by ID in ascending order using a lambda expression.

   i) Print the sorted list of employees.

   j) Sort employees by age in ascending order using a lambda expression.

   k) Print the sorted list of employees.

3. Implementation/Code:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

class Employee{
    private String name;
    private String id ;
    private int age;
    private double salary;
    public Employee(String name, String id, int age, double salary){
        this.name = name;
        this.id = id;
        this.age = age;
        this.salary = salary;
    }

    public String getName(){
        return name;
    }
    public String getId(){
        return id;
    }
    public int getAge(){
        return age;
    }
    public double getSalary(){
        return salary;
    }
    @Override
    public String toString() {
        return String.format("[ID: %s] %-10s | Age: %d | Salary: ₹%.2f", id, name, age,
salary);
    }
}

public class EmployeeSort {
    public static void main(String[] args) {
        List<Employee> emp = Arrays.asList(
                new Employee("Ansh Panwar", "11111", 26, 10000.0),
                new Employee("Naitik Raj", "2222", 21, 11000.0),
```

```
                new Employee("Sachin Singh Rathore", "33333", 22, 11000.0),
                new Employee("Shubham Pandey", "4444", 19, 12000.0)
        );

        emp.sort((e1, e2) -> Double.compare(e2.getSalary(), e1.getSalary()));
        System.out.println("Employees sorted by Salary:");
        emp.forEach(System.out::println);

        emp.sort((e1, e2) -> e1.getId().compareTo(e2.getId()));
        System.out.println("\nEmployees sorted by ID:");
        emp.forEach(System.out::println);

        emp.sort((e1, e2) -> Integer.compare(e1.getAge(), e2.getAge()));
        System.out.println("\nEmployees sorted by Age:");
        emp.forEach(System.out::println);

    }
}
```

4. **Output:**

```
 D:\PBLJ\E-6\out\production\E-6 EmployeeSort
Employees sorted by Salary:
[ID: 4444] Shubham Pandey | Age: 19 | Salary: ₹12000.00
[ID: 2222] Naitik Raj | Age: 21 | Salary: ₹11000.00
[ID: 33333] Sachin Singh Rathore | Age: 22 | Salary: ₹11000.00
[ID: 11111] Ansh Panwar | Age: 26 | Salary: ₹10000.00

Employees sorted by ID:
[ID: 11111] Ansh Panwar | Age: 26 | Salary: ₹10000.00
[ID: 2222] Naitik Raj | Age: 21 | Salary: ₹11000.00
[ID: 33333] Sachin Singh Rathore | Age: 22 | Salary: ₹11000.00
[ID: 4444] Shubham Pandey | Age: 19 | Salary: ₹12000.00

Employees sorted by Age:
[ID: 4444] Shubham Pandey | Age: 19 | Salary: ₹12000.00
[ID: 2222] Naitik Raj | Age: 21 | Salary: ₹11000.00
[ID: 33333] Sachin Singh Rathore | Age: 22 | Salary: ₹11000.00
[ID: 11111] Ansh Panwar | Age: 26 | Salary: ₹10000.00

Process finished with exit code 0
```

5. **Time Complexity: O(nlogn)**

6. **Space Complexity: O(1)**

7. **Learning Outcomes:**

i. Learned how to use **lambda expressions** to define custom sorting logic dynamically.
ii. **Understood sorting complexities and the importance of choosing efficient comparison strategies.**
iii. Improved code readability and efficiency using **functional programming concepts** in Java .

Experiment -2

1. **Aim** : - Create a program to use lambda expressions and stream operations to filter students scoring above 75%, sort them by marks, and display their names.

2. **Algorithm :-**
   a) **Define the `Student` class** with `name` and `marks`, a constructor for initialization, and getter methods. Override `toString()` for formatted output.

   b) **Create a list of `Student` objects** using `Arrays.asList()`, each with predefined names and marks.

   c) **Convert the list into a stream** to enable functional operations on the student data.

   d) **Filter students with marks above 75%** using the `filter()` method to retain only those meeting the criteria.

   e) **Sort the filtered students in descending order** using `sorted()` with `Double.compare(s2.getMarks(), s1.getMarks())`.

   f) **Extract student names** using `map(Student::getName)` and collect them into a `List<String>` using `collect()`.

   g) **Print the header message** "Students scoring above 75%" to introduce the output.

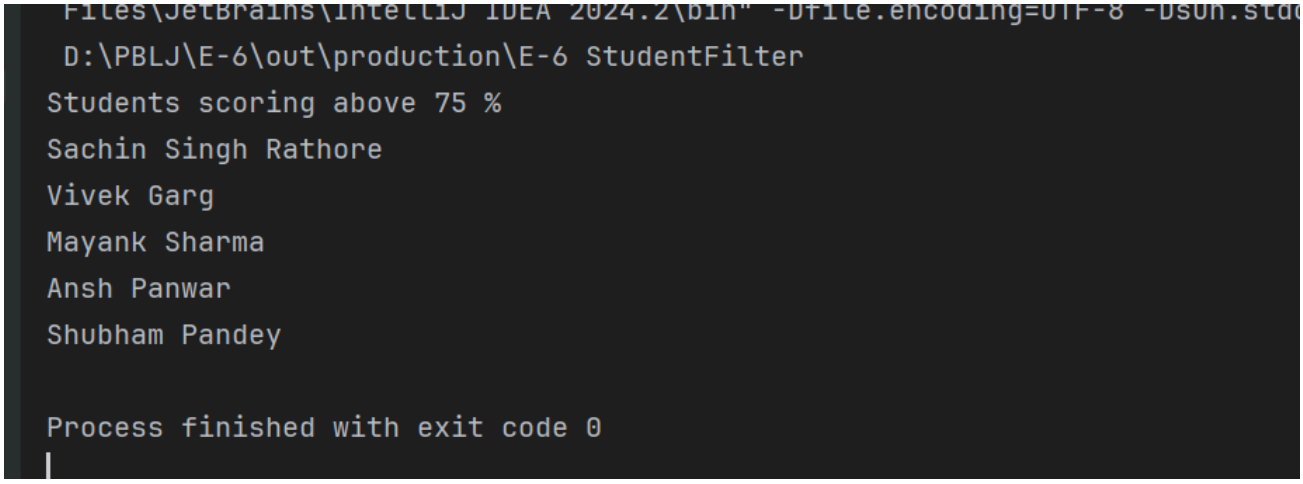
   h) **Iterate over the filtered names** using `forEach(System.out::println)` to display each student's name.

3. Code :-

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

class Student {
    private String name;
    private double marks;

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

    public String getName() { return name; }
    public double getMarks() { return marks; }

    @Override
    public String toString() {
        return String.format("%-20s Marks: %.2f%%", name, marks);
    }
}

public class StudentFilter {
    public static void main(String[] args) {
        List<Student> students = Arrays.asList(
                new Student("Ansh Panwar", 80.5),
                new Student("Naitik Raj", 72.3),
                new Student("Sachin Singh Rathore", 90.4),
                new Student("Shubham Pandey", 78.8),
                new Student("Mayank Sharma", 85.1),
                new Student("Kushan Jigyasu", 74.6),
                new Student("Vivek Garg", 88.2)
        );
```

```
        List<String>topStudents = students.stream().
                filter(s->s.getMarks() > 75).
                sorted((s1, s2) -> Double.compare(s2.getMarks(), s1.getMarks())).
                map(Student :: getName).
                collect(Collectors.toList());

        System.out.println("Students scoring above 75 %");
        topStudents.forEach(System.out::println);
    }
}
```

4. Output :-



5. Time Complexity :- O(nlogn)

6. Space Complexity :- O(n)

7. Learning Outcomes :-

a) **Understanding Streams in Java** – Learned how to use Java Streams for filtering, sorting, and mapping data efficiently.

b) **Filtering Data** – Gained experience in using the `filter()` method to extract specific elements based on conditions.

c) **Sorting Using Comparators** – Understood how to sort objects in descending order using `sorted()` with a custom comparator.

Experiment -3

1. **Aim :-** Write a Java program to process a large dataset of products using streams. Perform operations such as grouping products by category, finding the most expensive product in each category, and calculating the average price of all products.

2. ***Algorithm :-***

a) **Define the `Product` class** with attributes `name`, `category`, and `price`. Implement constructor, getters, and `toString()`.

b) **Create a list of `Product` objects** representing different categories and price ranges.

c) **Group products by category** using `Collectors.groupingBy()` and store them in a `TreeMap` for sorted categories.

d) **Find the most expensive product in each category** using `Collectors.maxBy()` with a price comparator.

e) **Calculate the average price of all products** using `mapToDouble()` and `average()`. Handle empty lists with `orElse(0.0)`.

f) **Display grouped products** by iterating through the `TreeMap` and printing each category with its products.

g) **Display the most expensive product in each category** using `Optional` to avoid null values.

h) **Print the average price of all products** formatted to two decimal places.

3. Code :-

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

class Product {
    private String name;
    private String category;
    private double price;

    public Product(String name, String category, double price) {
        this.name = name;
        this.category = category;
        this.price = price;
    }

    public String getName() { return name; }
    public String getCategory() { return category; }
    public double getPrice() { return price; }

    @Override
    public String toString() {
        return String.format("%-20s | ₹%.2f", name, price);
    }
}

public class ProductProcessor {
    public static void main(String[] args) {
        List<Product> products = Arrays.asList(
                new Product("iPhone 15", "Electronics", 79999.99),
                new Product("Samsung TV", "Electronics", 55999.50),
                new Product("Dell Laptop", "Electronics", 69999.00),
                new Product("Nike Shoes", "Fashion", 7999.99),
                new Product("Adidas Jacket", "Fashion", 4999.50),
                new Product("Rolex Watch", "Accessories", 250000.00),
                new Product("Gucci Sunglasses", "Accessories", 14999.75),
                new Product("Wooden Dining Table", "Furniture", 29999.99),
```

```
                new Product("Office Chair", "Furniture", 9999.00)
        );

        // Group products by category
        Map<String, List<Product>> groupedByCategory = products.stream()
                .collect(Collectors.groupingBy(Product::getCategory, TreeMap::new,
Collectors.toList()));

        Map<String, Optional<Product>> mostExpensiveByCategory = products.stream()
                .collect(Collectors.groupingBy(Product::getCategory,
Collectors.maxBy(Comparator.comparingDouble(Product::getPrice))));

        double avgPrice = products.stream()
                .mapToDouble(Product::getPrice)
                .average()
                .orElse(0.0);

        System.out.println("\n--- Product Categories ---");
        groupedByCategory.forEach((category, productList) -> {
            System.out.println("\nCategory: " + category);
            productList.forEach(System.out::println);
        });

        System.out.println("\n--- Most Expensive Product in Each Category ---");
        mostExpensiveByCategory.forEach((category, product) ->
                System.out.println(category + ": " + product.map(Product::toString).orElse("No
product"))
        );

        System.out.printf("\nAverage Price of All Products: ₹%.2f\n", avgPrice);
    }
}
```

4. **Output :-**

```
--- Product Categories ---

Category: Accessories
Rolex Watch           | ₹250000.00
Gucci Sunglasses      | ₹14999.75

Category: Electronics
iPhone 15             | ₹79999.99
Samsung TV            | ₹55999.50
Dell Laptop           | ₹69999.00

Category: Fashion
Nike Shoes            | ₹7999.99
Adidas Jacket         | ₹4999.50

Category: Furniture
Wooden Dining Table   | ₹29999.99
Office Chair          | ₹9999.00

--- Most Expensive Product in Each Category ---
Accessories: Rolex Watch           | ₹250000.00
```

5. **Time Complexity :- O(n)**

6. **Space Complexity :- O(n)**

7. **Learning Outcomes :-**
   a) **Efficient Data Processing with Java Streams** – Learned how to filter, group, and process data using Streams.

   b) **Grouping Data Dynamically** – Used `Collectors.groupingBy()` to categorize objects efficiently.

   c) **Finding Maximum Values in Groups** – Used `Collectors.maxBy()` to identify the highest-priced item per category.