Experiment 6

Student Name: *Madhav* **UID:22BCS50195**
Branch: CSE **Section/Group:641-B**
Semester: 6th **Date of Performance:25-02-2025**
Subject Name: Java with Lab **Subject Code: 22CSH-359**

1. Aim:
Develop Java programs using lambda expressions and stream operations for sorting, filtering, and processing large datasets efficiently.

2. Objective:

- Develop Java programs using lambda expressions and stream operations for sorting, filtering, and processing large datasets efficiently.
- Implement easy, medium, and hard-level tasks involving sorting employees, filtering and sorting students, and processing products using streams.

3. Implementation/Code:

a.
```
import java.util.*;
class Employee {
   String name;
   int age;
   double salary;
   Employee(String name, int age, double salary) {
      this.name = name;
      this.age = age;
      this.salary = salary;
   }
   @Override
   public String toString() {
      return name + " - Age: " + age + ", Salary: " + salary;
   }
}
public class EmployeeSort {
   public static void main(String[] args) {
      List<Employee> employees = Arrays.asList(
         new Employee("Madhav", 10, 60000),
         new Employee("vinod", 20, 100000),
```

```
            new Employee("tejas", 30, 70000)
        );
        employees.sort(Comparator.comparing(emp -> emp.name));
        System.out.println("Sorted by Name: " + employees);
        employees.sort(Comparator.comparingInt(emp -> emp.age));
        System.out.println("Sorted by Age: " + employees);
        employees.sort(Comparator.comparingDouble(emp -> emp.salary));
        System.out.println("Sorted by Salary: " + employees);
    }
}
```

b.

```
import java.util.*;
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 name + " - Marks: " + marks;
    }
}

public class StudentFilter {
    public static void main(String[] args) {
        List<Student> students = Arrays.asList(
            new Student("Madhav", 88),
            new Student("Saurav", 72),
            new Student("Tejas", 95),
            new Student("Nikhil", 68),
            new Student("Ishaan", 79)
        );
```

```
        List<String> topStudents = students.stream()
                .filter(s -> s.getMarks() > 75)
                .sorted(Comparator.comparingDouble(Student::getMarks).reversed())
                .map(Student::getName)
                .collect(Collectors.toList());

        System.out.println("\nTop Students: " + topStudents);
    }
     }
```

C)

```
import java.util.*;
import java.util.stream.Collectors;
class Product {
    String name;
    String category;
    double price;
    public Product(String name, String category, double price) {
        this.name = name;
        this.category = category;
        this.price = price;
    }
    @Override
    public String toString() {
        return name + " ($" + price + ")";
    }
}
public class ProductProcessor {
    public static void main(String[] args) {
        List<Product> products = Arrays.asList(
            new Product("Laptop", "Electronics", 1200),
            new Product("Phone", "Electronics", 800),
            new Product("TV", "Electronics", 1500),
            new Product("Shirt", "Clothing", 50),
            new Product("Jeans", "Clothing", 70),
            new Product("Blender", "Appliances", 200),
            new Product("Toaster", "Appliances", 100)
        );
```

```
        Map<String, List<Product>> productsByCategory = products.stream()
            .collect(Collectors.groupingBy(p -> p.category));
        System.out.println("Products grouped by category:");
        productsByCategory.forEach((category, productList) ->
            System.out.println(category + ": " + productList));
        Map<String, Optional<Product>> mostExpensiveByCategory = products.stream()
            .collect(Collectors.groupingBy(
                p -> p.category,
                Collectors.maxBy(Comparator.comparingDouble(p -> p.price))
            ));
        System.out.println("\nMost expensive product in each category:");
        mostExpensiveByCategory.forEach((category, product) ->
            System.out.println(category + ": " + product.orElse(null)));
        double averagePrice = products.stream()
            .mapToDouble(p -> p.price)
            .average()
            .orElse(0);
        System.out.println("\nAverage price of all products: $" + averagePrice);
    }
}
```

4. Output:

A) Sorted by Name:

Madhav - Age: 10, Salary: $60000.0
Tejas - Age: 30, Salary: $70000.0
Vinod - Age: 20, Salary: $100000.0

Sorted by Age:

Madhav - Age: 10, Salary: $60000.0
Vinod - Age: 20, Salary: $100000.0
Tejas - Age: 30, Salary: $70000.0

Sorted by Salary:

Madhav - Age: 10, Salary: $60000.0
Tejas - Age: 30, Salary: $70000.0
Vinod - Age: 20, Salary: $100000.0

B) Top Students: [Tejas, Madhav, Ishaan]

C) Products grouped by category:
Electronics: [Tablet ($900), Smartwatch ($350), Gaming Console ($700)]
Footwear: [Sneakers ($120)]
Clothing: [Jacket ($150)]
Appliances: [Microwave ($250), Air Fryer ($180)]

Most expensive product in each category:
Electronics: Tablet ($900)
Footwear: Sneakers ($120)
Clothing: Jacket ($150)
Appliances: Microwave ($250)

Average price of all products: $378.57

5. Learning Outcome:

- Understand and implement **lambda expressions** for sorting objects in a list based on different attributes.
- Utilize **Java Streams API** to perform operations like **filtering, sorting, and mapping** efficiently on large datasets.
- Learn **Comparator and method references** to simplify object comparisons for sorting.
- Apply **grouping and aggregation functions** using Collectors.groupingBy() and Collectors.maxBy() for processing categorized data.
- Gain hands-on experience in computing **statistical values** like the **average** from a dataset using mapToDouble() and average().
- Improve **code efficiency and readability** by using **functional programming** techniques in Java.