Experiment 6.1

Student Name: Akshat Srivastava **UID:** 22BCS11740
Branch: BE CSE **Section/Group:** 22BCS_IOT_618_A
Semester: 6th **DoP:** 28/02/2025
Subject Name: PBLJ Lab **Subject Code:** 22CSH-359

1. **Aim:** To implement a Java program that sorts a list of Employee objects based on name, age, and salary using lambda expressions and stream operations.

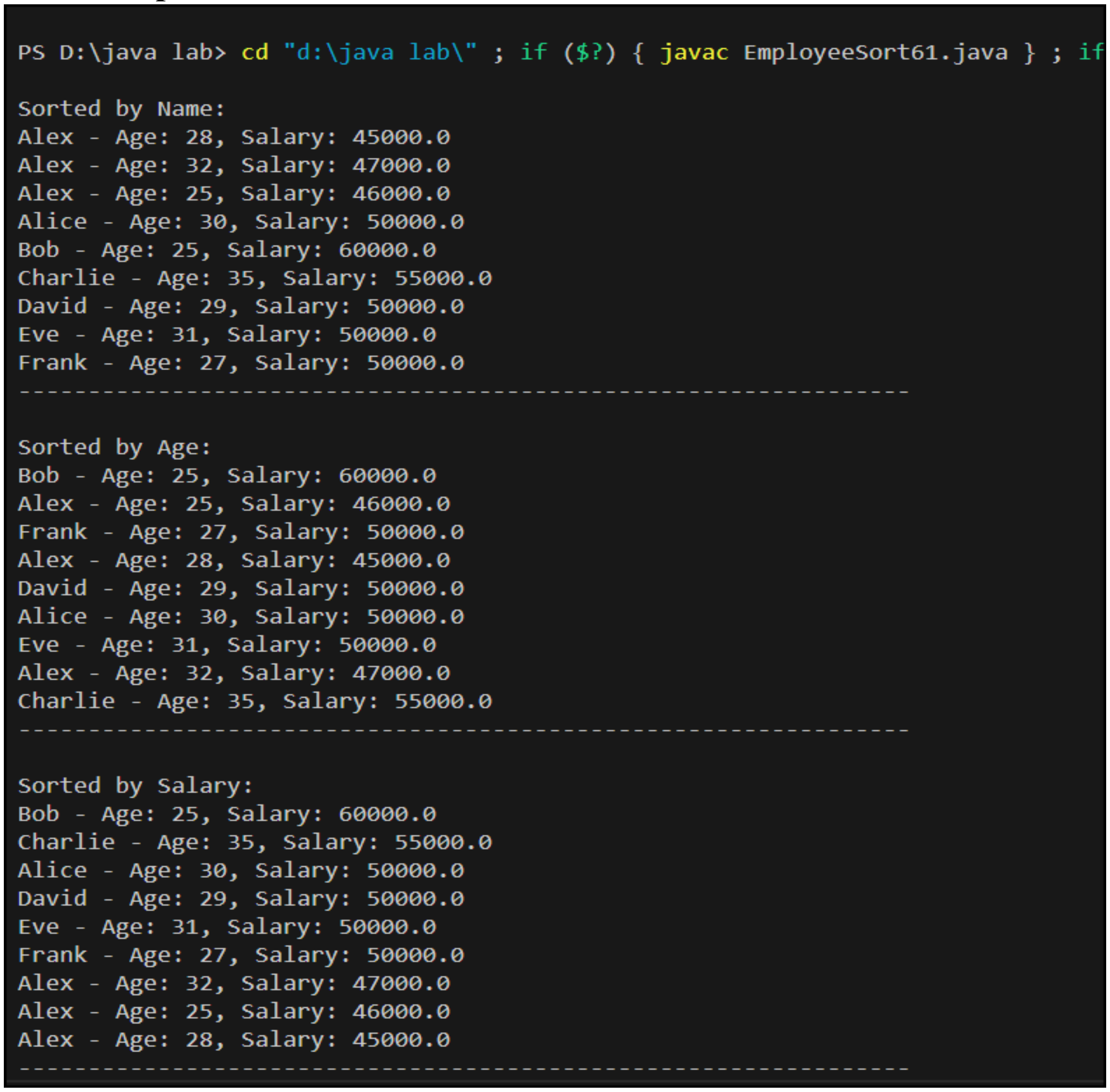
2. **Objective:**

- Understand the use of lambda expressions for sorting objects.
- Learn how to use streams for efficient data processing.
- Implement Comparator with lambda for different sorting criteria.
- Use method references to display sorted employee details.

3. **Implementation/Code:**

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

class Employee {
    String name;
    int age;
    double salary;

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

    void display() {
        System.out.println(name + " - Age: " + age + ", Salary: " + salary);
    }
}

public class EmployeeSort61 {
    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(
            new Employee("Alice", 30, 50000),
            new Employee("Bob", 25, 60000),
            new Employee("Charlie", 35, 55000),
            new Employee("Alex", 28, 45000),
            new Employee("Alex", 32, 47000),
            new Employee("Alex", 25, 46000),
            new Employee("David", 29, 50000),
            new Employee("Eve", 31, 50000),
            new Employee("Frank", 27, 50000)
        );

        System.out.println("\nSorted by Name:");
        employees.stream()
                .sorted(Comparator.comparing(e -> e.name))
                .forEach(Employee::display);
        System.out.println("--------------------------------------------------------------");

        System.out.println("\nSorted by Age:");
        employees.stream()
                .sorted(Comparator.comparingInt(e -> e.age))
                .forEach(Employee::display);
        System.out.println("--------------------------------------------------------------");
        System.out.println("\nSorted by Salary:");
        employees.stream()
                .sorted(Comparator.comparingDouble(e -> -e.salary))
                .forEach(Employee::display);
        System.out.println("------------------------------------------------------------");

    }
}
```

4. Output



```
PS D:\java lab> cd "d:\java lab\" ; if ($?) { javac EmployeeSort61.java } ; if

Sorted by Name:
Alex - Age: 28, Salary: 45000.0
Alex - Age: 32, Salary: 47000.0
Alex - Age: 25, Salary: 46000.0
Alice - Age: 30, Salary: 50000.0
Bob - Age: 25, Salary: 60000.0
Charlie - Age: 35, Salary: 55000.0
David - Age: 29, Salary: 50000.0
Eve - Age: 31, Salary: 50000.0
Frank - Age: 27, Salary: 50000.0
----------------------------------------------------------------

Sorted by Age:
Bob - Age: 25, Salary: 60000.0
Alex - Age: 25, Salary: 46000.0
Frank - Age: 27, Salary: 50000.0
Alex - Age: 28, Salary: 45000.0
David - Age: 29, Salary: 50000.0
Alice - Age: 30, Salary: 50000.0
Eve - Age: 31, Salary: 50000.0
Alex - Age: 32, Salary: 47000.0
Charlie - Age: 35, Salary: 55000.0
----------------------------------------------------------------

Sorted by Salary:
Bob - Age: 25, Salary: 60000.0
Charlie - Age: 35, Salary: 55000.0
Alice - Age: 30, Salary: 50000.0
David - Age: 29, Salary: 50000.0
Eve - Age: 31, Salary: 50000.0
Frank - Age: 27, Salary: 50000.0
Alex - Age: 32, Salary: 47000.0
Alex - Age: 25, Salary: 46000.0
Alex - Age: 28, Salary: 45000.0
----------------------------------------------------------------
```

5. Learning Outcome:

- Gain hands-on experience with lambda expressions and functional programming in Java.
- Learn how to efficiently sort objects using Java Streams.
- Understand how to use Comparator.comparing() for custom sorting.
- Learn how to use forEach() with method references to print sorted lists.

Experiment 6.2

Student Name: Akshat Srivastava **UID:** 22BCS11740
Branch: BE CSE **Section/Group:** 22BCS_IOT_618_A
Semester: 6th **DoP:** 28/02/2025
Subject Name: PBLJ Lab **Subject Code:** 22CSH-359

1. **Aim:** To implement a Java program using lambda expressions and the Stream API to filter and sort students based on their marks.

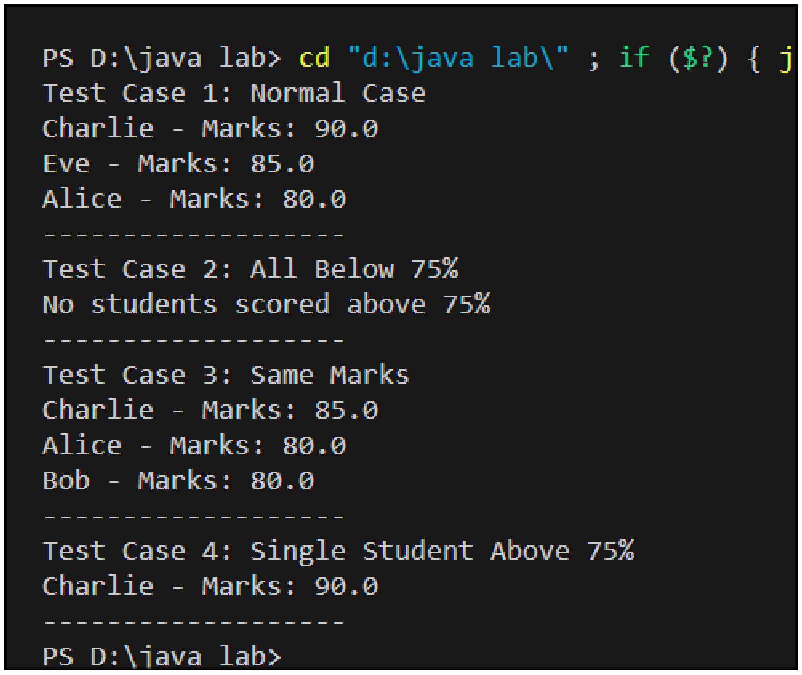
2. **Objective:**

   - Understand lambda expressions for concise code.
   - Learn how to filter data using the Stream API.
   - Implement sorting using comparators in streams.
   - Display filtered and sorted student data using method references.

3. **Implementation/Code:**

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

class Student {
    String name;
    double marks;

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

    double getMarks() {
        return marks;
    }

    void display() {
        System.out.println(name + " - Marks: " + marks);
    }
}

public class StudentFilterSort62 {
    public static void main(String[] args) {
        testCase1();
        testCase2();
        testCase3();
        testCase4();
    }

    static void processAndDisplay(List<Student> students) {
        List<Student> filteredSortedStudents = students.stream()
            .filter(s -> s.getMarks() > 75)
            .sorted(Comparator.comparingDouble(Student::getMarks).reversed()
                .thenComparing(Student::getName))
            .collect(Collectors.toList());

        if (filteredSortedStudents.isEmpty()) {
            System.out.println("No students scored above 75%");
        } else {
            filteredSortedStudents.forEach(Student::display);
        }
        System.out.println("------------------");
    }

    static void testCase1() {
        System.out.println("Test Case 1: Normal Case");
        List<Student> students = Arrays.asList(
            new Student("Alice", 80),
            new Student("Bob", 72),
            new Student("Charlie", 90),
            new Student("David", 65),
            new Student("Eve", 85)
        );
        processAndDisplay(students);
    }

    static void testCase2() {
        System.out.println("Test Case 2: All Below 75%");
        List<Student> students = Arrays.asList(
            new Student("Bob", 70),
            new Student("David", 60),
            new Student("Frank", 65)
        );
        processAndDisplay(students);
    }

    static void testCase3() {
        System.out.println("Test Case 3: Same Marks");
        List<Student> students = Arrays.asList(
            new Student("Alice", 80),
            new Student("Bob", 80),
            new Student("Charlie", 85)
        );
        processAndDisplay(students);
    }
    static void testCase4() {
        System.out.println("Test Case 4: Single Student Above 75%");
        List<Student> students = Arrays.asList(
            new Student("Alice", 60),
            new Student("Bob", 50),
            new Student("Charlie", 90)
        );
        processAndDisplay(students);
    }
}
```

4. Output



```
PS D:\java lab> cd "d:\java lab\" ; if ($?) { j
Test Case 1: Normal Case
Charlie - Marks: 90.0
Eve - Marks: 85.0
Alice - Marks: 80.0
-------------------
Test Case 2: All Below 75%
No students scored above 75%
-------------------
Test Case 3: Same Marks
Charlie - Marks: 85.0
Alice - Marks: 80.0
Bob - Marks: 80.0
-------------------
Test Case 4: Single Student Above 75%
Charlie - Marks: 90.0
-------------------
PS D:\java lab>
```

5. Learning Outcome:

- Ability to use Streams to process collections efficiently.
- Understand how to filter elements based on a condition.
- Learn how to sort data in descending order using Streams.
- Gain proficiency in lambda expressions and method references.

Experiment 6.3

Student Name: Akshat Srivastava **UID:** 22BCS11740
Branch: BE CSE **Section/Group:** 22BCS_IOT_618_A
Semester: 6th **DoP:** 28/02/2025
Subject Name: PBLJ Lab **Subject Code:** 22CSH-359

1. **Aim:** To demonstrate grouping, finding max values, and calculating averages with Java Streams.

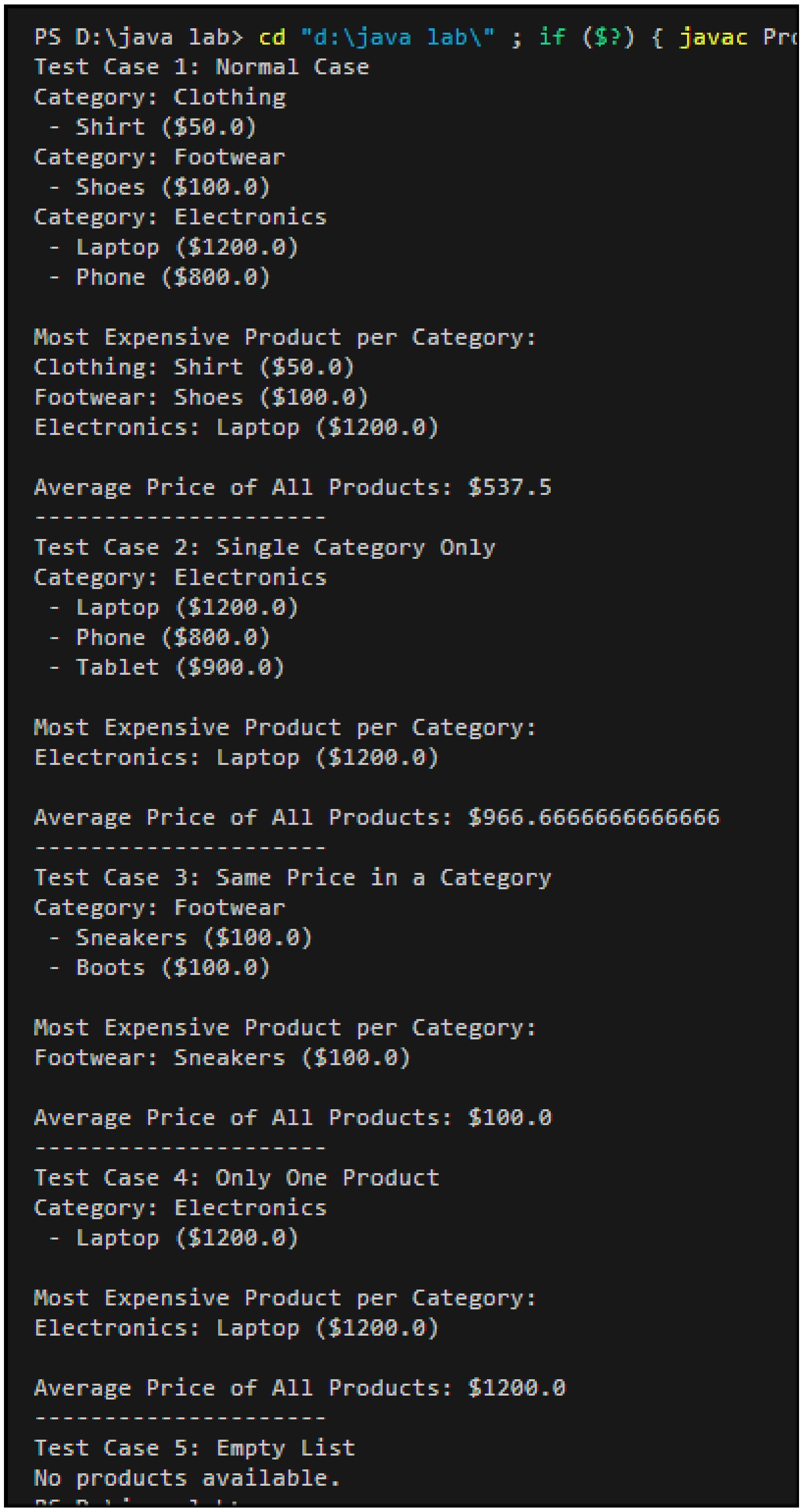
2. **Objective:**

   - Understand how to group products by category.
   - Use Streams API for efficient filtering and aggregation.
   - Find the most expensive product in each category.
   - Compute the average price of all products.
   - Learn to handle edge cases in data processing.

3. **Implementation/Code:**

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

class Product {
    String name, category;
    double price;

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

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

public class ProductProcessor63 {
    public static void main(String[] args) {
        testCase1();
        testCase2();
        testCase3();
        testCase4();
        testCase5();
    }

    static void processAndDisplay(List<Product> products) {
        if (products.isEmpty()) {
            System.out.println("No products available.");
            return;
        }

        Map<String, List<Product>> groupedByCategory = products.stream()
            .collect(Collectors.groupingBy(Product::getCategory));

        Map<String, Optional<Product>> mostExpensiveByCategory =
products.stream()
            .collect(Collectors.groupingBy(Product::getCategory,

Collectors.maxBy(Comparator.comparingDouble(Product::getPrice))));

        double avgPrice = products.stream()
            .collect(Collectors.averagingDouble(Product::getPrice));
        groupedByCategory.forEach((category, list) -> {
            System.out.println("Category: " + category);
            list.forEach(p -> System.out.println(" - " + p.name + " ($" + p.price +
")"));
        });

        System.out.println("\nMost Expensive Product per Category:");
        mostExpensiveByCategory.forEach((category, product) ->
            System.out.println(category + ": " + product.get().name + " ($" +
product.get().price + ")"));

        System.out.println("\nAverage Price of All Products: $" + avgPrice);
        System.out.println("--------------------");
    }

    static void testCase1() {
        System.out.println("Test Case 1: Normal Case");
        List<Product> products = Arrays.asList(
            new Product("Laptop", "Electronics", 1200),
            new Product("Phone", "Electronics", 800),
            new Product("Shirt", "Clothing", 50),
            new Product("Shoes", "Footwear", 100)
        );
        processAndDisplay(products);
    }

    static void testCase2() {
        System.out.println("Test Case 2: Single Category Only");
        List<Product> products = Arrays.asList(
            new Product("Laptop", "Electronics", 1200),
            new Product("Phone", "Electronics", 800),
            new Product("Tablet", "Electronics", 900)
        );
        processAndDisplay(products);
    }

    static void testCase3() {
        System.out.println("Test Case 3: Same Price in a Category");
        List<Product> products = Arrays.asList(
            new Product("Sneakers", "Footwear", 100),
            new Product("Boots", "Footwear", 100)
        );
        processAndDisplay(products);
    }

    static void testCase4() {
        System.out.println("Test Case 4: Only One Product");
        List<Product> products = Collections.singletonList(
            new Product("Laptop", "Electronics", 1200)
        );
        processAndDisplay(products);
    }

    static void testCase5() {
        System.out.println("Test Case 5: Empty List");
        List<Product> products = new ArrayList<>();
        processAndDisplay(products);
    }
}
```

4. **Output**



```
PS D:\java lab> cd "d:\java lab\" ; if ($?) { javac Pro
Test Case 1: Normal Case
Category: Clothing
 - Shirt ($50.0)
Category: Footwear
 - Shoes ($100.0)
Category: Electronics
 - Laptop ($1200.0)
 - Phone ($800.0)

Most Expensive Product per Category:
Clothing: Shirt ($50.0)
Footwear: Shoes ($100.0)
Electronics: Laptop ($1200.0)

Average Price of All Products: $537.5
---------------------
Test Case 2: Single Category Only
Category: Electronics
 - Laptop ($1200.0)
 - Phone ($800.0)
 - Tablet ($900.0)

Most Expensive Product per Category:
Electronics: Laptop ($1200.0)

Average Price of All Products: $966.6666666666666
---------------------
Test Case 3: Same Price in a Category
Category: Footwear
 - Sneakers ($100.0)
 - Boots ($100.0)

Most Expensive Product per Category:
Footwear: Sneakers ($100.0)

Average Price of All Products: $100.0
---------------------
Test Case 4: Only One Product
Category: Electronics
 - Laptop ($1200.0)

Most Expensive Product per Category:
Electronics: Laptop ($1200.0)

Average Price of All Products: $1200.0
---------------------
Test Case 5: Empty List
No products available.
```

5. Learning Outcome:

- Ability to use Collectors.groupingBy() for categorization.
- Use Collectors.maxBy() to find the highest-priced product per category.
- Apply Collectors.averagingDouble() for price calculations.