Experiment 6

Student Name: Sarthak Rana **UID:22BCS16222**
Branch: BE-CSE **Section/Group:642/B**
Semester:6th **Date of Performance:**
Subject Name: Project Based Learning in Java with Lab **Subject Code: 22CSH-359**

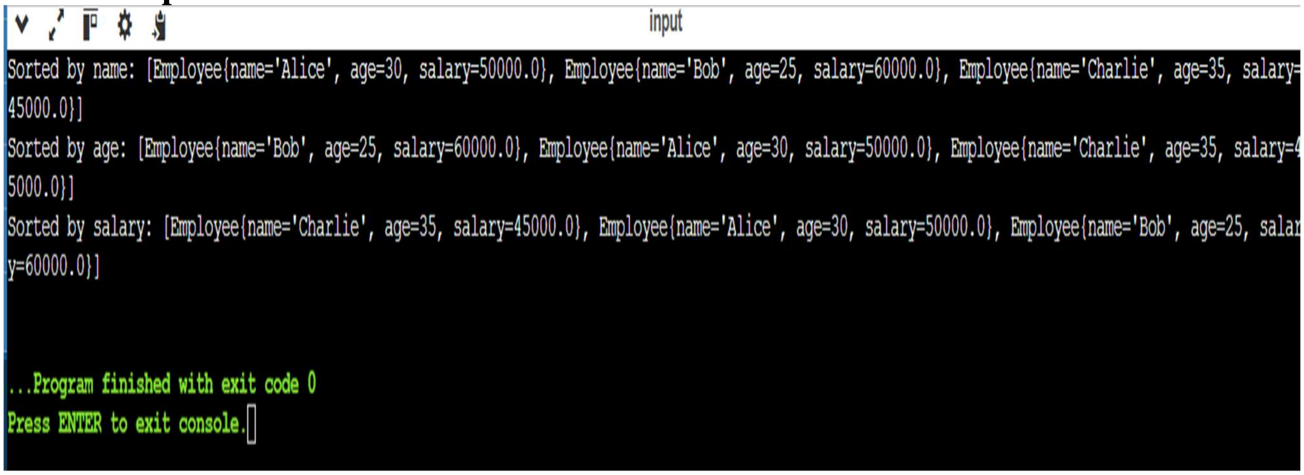
1. **Aim:** To implement a Java program that sorts a list of Employee objects (based on name, age, and salary) using lambda expressions and stream operations to demonstrate efficient data processing**.**

2. Implementation/Code:

```
import java.util.*;

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

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

   @Override
   public String toString() {
      return "Employee{name='" + name + "', age=" + age + ", salary=" + salary + "}";
   }
}

public class EmployeeSort {
   public static void main(String[] args) {
      List<Employee> employees = new ArrayList<>();
      employees.add(new Employee("Alice", 30, 50000));
      employees.add(new Employee("Bob", 25, 60000));
      employees.add(new Employee("Charlie", 35, 45000));

      // Sort by name
      employees.sort((e1, e2) -> e1.name.compareTo(e2.name));
      System.out.println("Sorted by name: " + employees);

      // Sort by age
```

```
        employees.sort((e1, e2) -> Integer.compare(e1.age, e2.age));
        System.out.println("Sorted by age: " + employees);

        // Sort by salary
        employees.sort((e1, e2) -> Double.compare(e1.salary, e2.salary));
        System.out.println("Sorted by salary: " + employees);
    }
}
```

3. Output:



```
Sorted by name: [Employee{name='Alice', age=30, salary=50000.0}, Employee{name='Bob', age=25, salary=60000.0}, Employee{name='Charlie', age=35, salary=
45000.0}]
Sorted by age: [Employee{name='Bob', age=25, salary=60000.0}, Employee{name='Alice', age=30, salary=50000.0}, Employee{name='Charlie', age=35, salary=4
5000.0}]
Sorted by salary: [Employee{name='Charlie', age=35, salary=45000.0}, Employee{name='Alice', age=30, salary=50000.0}, Employee{name='Bob', age=25, salar
y=60000.0}]


...Program finished with exit code 0
Press ENTER to exit console.
```

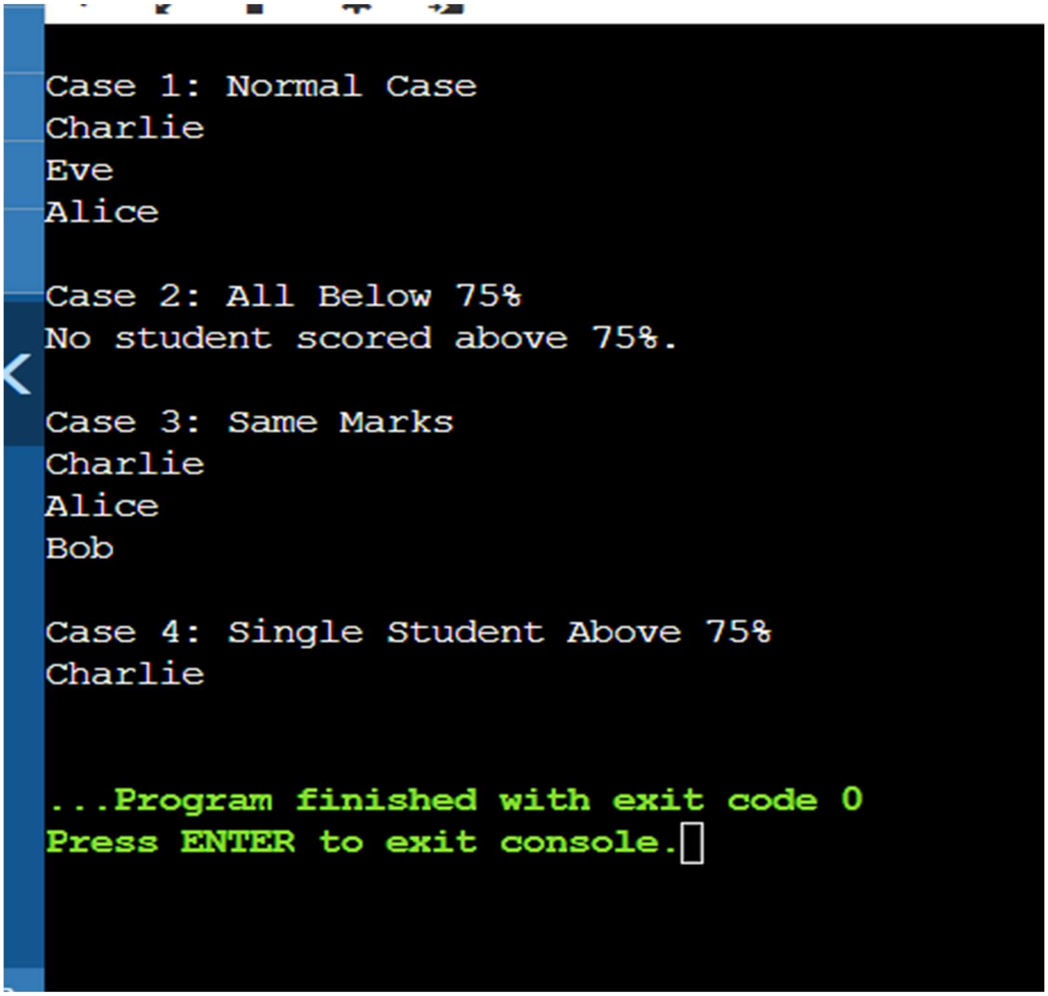
Experiment 6.2

Aim: Implement Java program that uses lambda expressions and Stream API to filter students who scored above 75%, sort them by marks, and display their names.

Code:

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

class Student {
    String name;
    double marks;

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

    public void display() {
        System.out.println(name);
    }
}

public class StudentFilterSort {
    public static void main(String[] args) {
        runTestCase("Case 1: Normal Case", Arrays.asList(
            new Student("Alice", 80),
            new Student("Bob", 72),
            new Student("Charlie", 90),
            new Student("David", 65),
            new Student("Eve", 85)
        ));

        runTestCase("Case 2: All Below 75%", Arrays.asList(
            new Student("Bob", 70),
            new Student("David", 60),
            new Student("Frank", 65)
        ));

        runTestCase("Case 3: Same Marks", Arrays.asList(
            new Student("Alice", 80),
            new Student("Bob", 80),
            new Student("Charlie", 85)
        ));

        runTestCase("Case 4: Single Student Above 75%", Arrays.asList(
            new Student("Alice", 60),
            new Student("Bob", 50),
            new Student("Charlie", 90)
        ));
    }

    private static void runTestCase(String caseName, List<Student> students) {
        System.out.println("\n" + caseName);

        List<Student> filteredSortedStudents = students.stream()
            .filter(s -> s.marks > 75) // Filter students with marks > 75
            .sorted((s1, s2) -> {
                int markComparison = Double.compare(s2.marks, s1.marks);
                return markComparison != 0 ? markComparison : s1.name.compareTo(s2.name);
            }) // Sort by marks descending, then by name ascending
            .collect(Collectors.toList()); // Collect into a new list

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

Output:



```
Case 1: Normal Case
Charlie
Eve
Alice

Case 2: All Below 75%
No student scored above 75%.

Case 3: Same Marks
Charlie
Alice
Bob

Case 4: Single Student Above 75%
Charlie


...Program finished with exit code 0
Press ENTER to exit console.
```

Experiment 6.3

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.

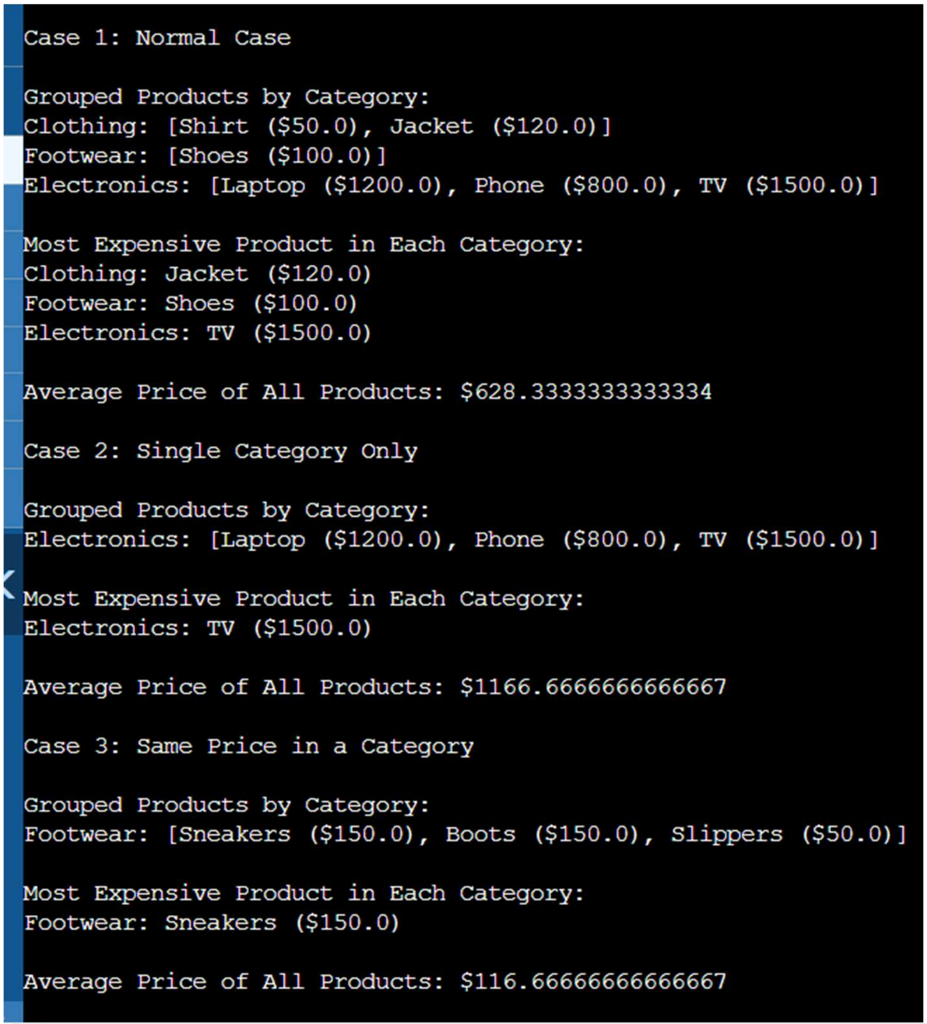
Code:

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

class Product {
   String name;
   String category;
```

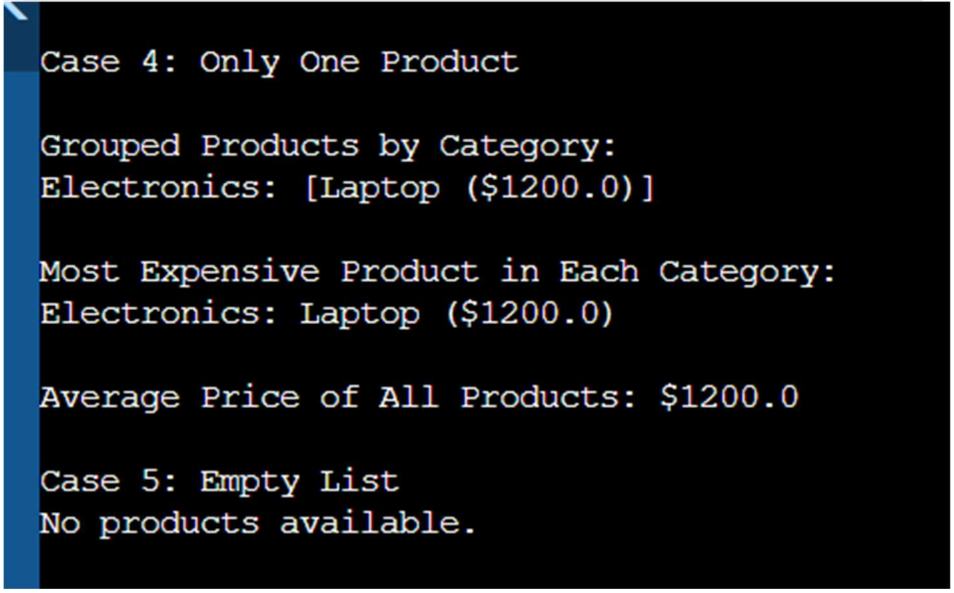
```
    double price;

    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) {
        runTestCase("Case 1: Normal Case", Arrays.asList(
            new Product("Laptop", "Electronics", 1200),
            new Product("Phone", "Electronics", 800),
            new Product("Shirt", "Clothing", 50),
            new Product("Shoes", "Footwear", 100),
            new Product("TV", "Electronics", 1500),
            new Product("Jacket", "Clothing", 120)
        ));

        runTestCase("Case 2: Single Category Only", Arrays.asList(
            new Product("Laptop", "Electronics", 1200),
            new Product("Phone", "Electronics", 800),
            new Product("TV", "Electronics", 1500)
        ));

        runTestCase("Case 3: Same Price in a Category", Arrays.asList(
            new Product("Sneakers", "Footwear", 150),
            new Product("Boots", "Footwear", 150),
            new Product("Slippers", "Footwear", 50)
        ));

        runTestCase("Case 4: Only One Product", Arrays.asList(
            new Product("Laptop", "Electronics", 1200)
        ));

        runTestCase("Case 5: Empty List", new ArrayList<>());
    }

    private static void runTestCase(String caseName, List<Product> products) {
        System.out.println("\n" + caseName);

        // Grouping products by category
        Map<String, List<Product>> groupedByCategory = products.stream()
```

```
            .collect(Collectors.groupingBy(p -> p.category));

        // Finding the most expensive product in each category
        Map<String, Optional<Product>> mostExpensiveByCategory = products.stream()
            .collect(Collectors.groupingBy(
                p -> p.category,
                Collectors.maxBy(Comparator.comparingDouble(p -> p.price))
            ));

        // Calculating the average price of all products
        double averagePrice = products.stream()
            .collect(Collectors.averagingDouble(p -> p.price));

        // Display grouped products
        if (groupedByCategory.isEmpty()) {
            System.out.println("No products available.");
        } else {
            System.out.println("\nGrouped Products by Category:");
            groupedByCategory.forEach((category, productList) ->
                System.out.println(category + ": " + productList)
            );

            // Display most expensive product in each category
            System.out.println("\nMost Expensive Product in Each Category:");
            mostExpensiveByCategory.forEach((category, product) ->
                System.out.println(category + ": " + product.orElse(null))
            );

            // Display average price of all products
            System.out.println("\nAverage Price of All Products: $" + averagePrice);
        }
    }
}
```

Output:



```
Case 1: Normal Case

Grouped Products by Category:
Clothing: [Shirt ($50.0), Jacket ($120.0)]
Footwear: [Shoes ($100.0)]
Electronics: [Laptop ($1200.0), Phone ($800.0), TV ($1500.0)]

Most Expensive Product in Each Category:
Clothing: Jacket ($120.0)
Footwear: Shoes ($100.0)
Electronics: TV ($1500.0)

Average Price of All Products: $628.3333333333334

Case 2: Single Category Only

Grouped Products by Category:
Electronics: [Laptop ($1200.0), Phone ($800.0), TV ($1500.0)]

Most Expensive Product in Each Category:
Electronics: TV ($1500.0)

Average Price of All Products: $1166.6666666666667

Case 3: Same Price in a Category

Grouped Products by Category:
Footwear: [Sneakers ($150.0), Boots ($150.0), Slippers ($50.0)]

Most Expensive Product in Each Category:
Footwear: Sneakers ($150.0)

Average Price of All Products: $116.66666666666667
```



```
Case 4: Only One Product

Grouped Products by Category:
Electronics: [Laptop ($1200.0)]

Most Expensive Product in Each Category:
Electronics: Laptop ($1200.0)

Average Price of All Products: $1200.0

Case 5: Empty List
No products available.
```