Experiment 6

Student Name: Khushi	**UID: 22BCS15811**
Branch: BE-CSE	**Section/Group: 618_B**
Semester: 6	**Date of Performance:28/2/25**
Subject Name:PBLJ	**Subject Code: 22CSH-359**

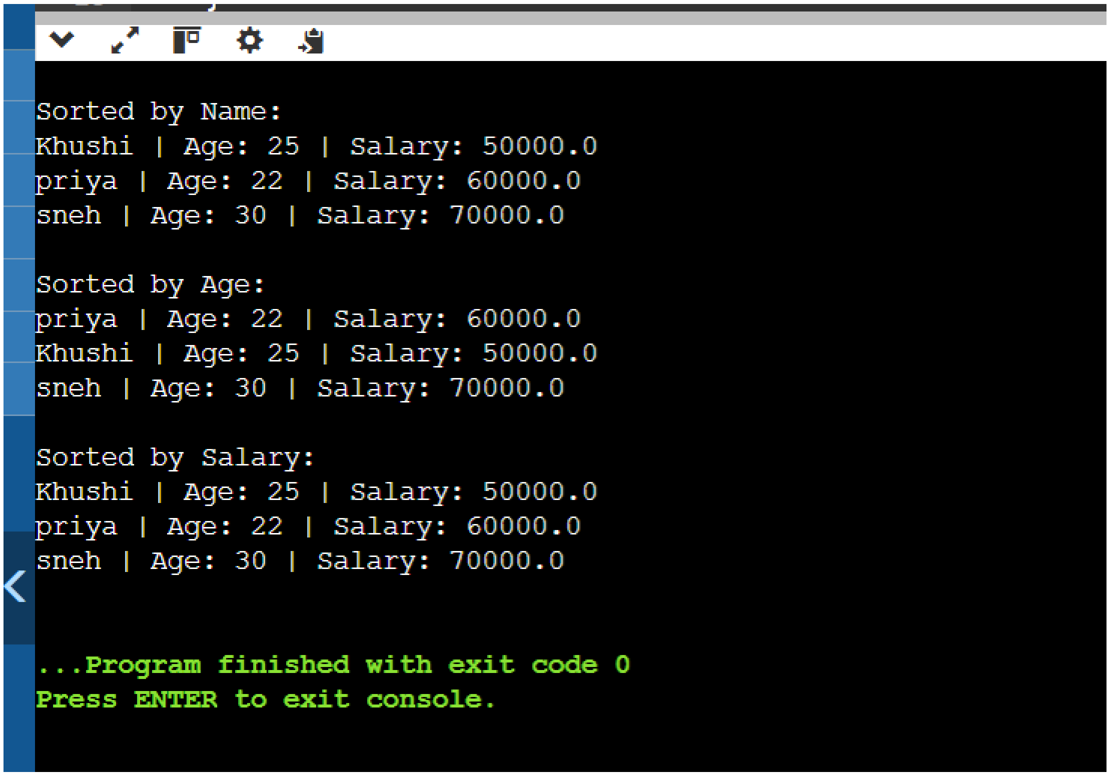
Problem 1 : Write a program to sort a list of Employee objects using lambda expressions**.**

1. **Objective:** The program sorts employees by name, age, and salary using Collections.sort() with concise, readable lambda functions. This helps in understanding how functional programming concepts simplify sorting in Java.

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;
    }
    public void display() {
        System.out.println(name + " | Age: " + age + " | Salary: " + salary);
    }
}
public class EmployeeSort {
    public static void main(String[] args) {
        List<Employee> employees = new ArrayList<>();
        employees.add(new Employee("Khushi", 25, 50000));
        employees.add(new Employee("sneh", 30, 70000));
        employees.add(new Employee("priya", 22, 60000));
        employees.sort((e1, e2) -> e1.name.compareTo(e2.name));
        System.out.println("Sorted by Name:");
        employees.forEach(Employee::display);
        employees.sort((e1, e2) -> Integer.compare(e1.age, e2.age));
        System.out.println("Sorted by Age:");
        employees.forEach(Employee::display);
        employees.sort((e1, e2) -> Double.compare(e1.salary, e2.salary));
        System.out.println("Sorted by Salary:");
        employees.forEach(Employee::display);
    }}
```

3. Output



```
Sorted by Name:
Khushi | Age: 25 | Salary: 50000.0
priya | Age: 22 | Salary: 60000.0
sneh | Age: 30 | Salary: 70000.0

Sorted by Age:
priya | Age: 22 | Salary: 60000.0
Khushi | Age: 25 | Salary: 50000.0
sneh | Age: 30 | Salary: 70000.0

Sorted by Salary:
Khushi | Age: 25 | Salary: 50000.0
priya | Age: 22 | Salary: 60000.0
sneh | Age: 30 | Salary: 70000.0


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

Problem 2: Create a program to use lambda expressions and stream operations to filter students scoring above 75%, sort them by marks, and display their names.

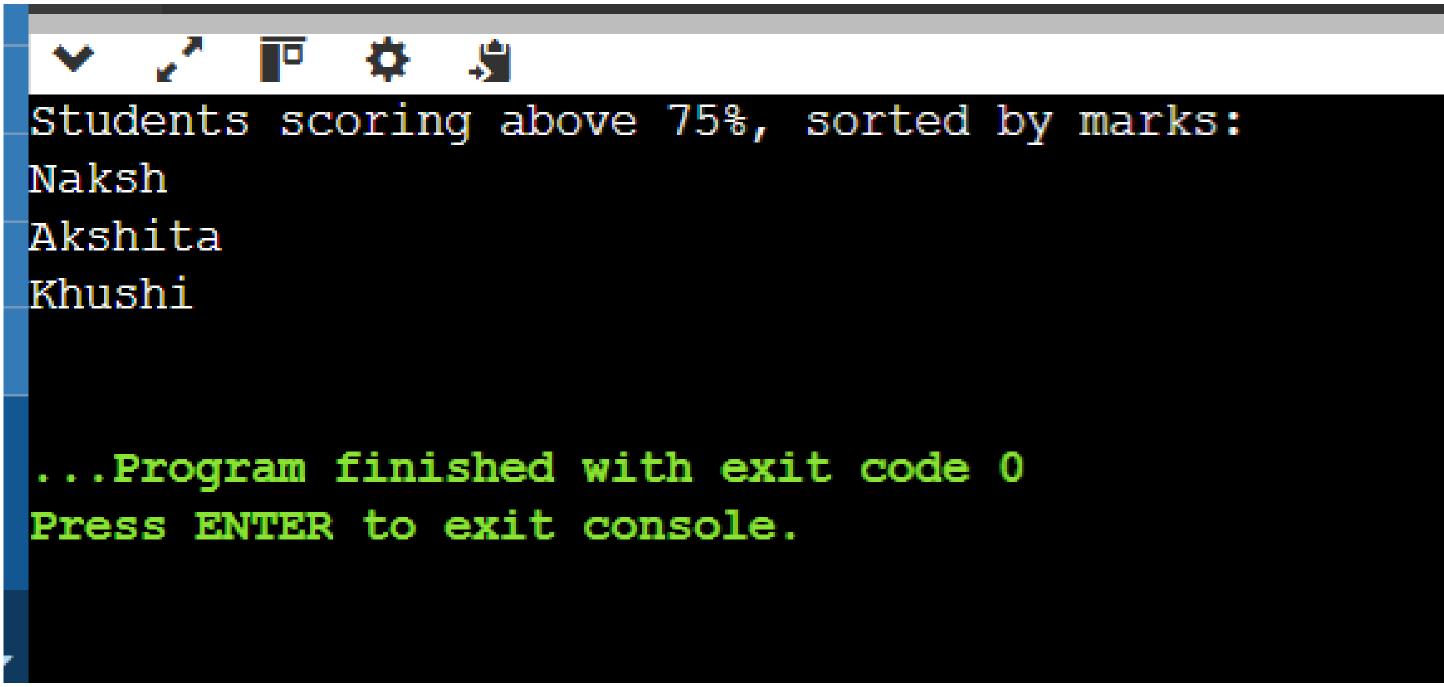
Objective:

The program filters students who have scored above 75%, ensuring only high-achieving students are considered. It then sorts the filtered students in descending order based on their marks to highlight the top performers. Finally, it extracts and displays their names in a structured format.

CODE:

```
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;
    }
}
public class StudentFilter {
    public static void main(String[] args) {
        List<Student> students = Arrays.asList(
            new Student("Khushi", 80),
            new Student("Koml", 70),
            new Student("Akshita", 85),
            new Student("Priya", 60),
            new Student("Naksh", 90)
        );
        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("Students scoring above 75%, sorted by marks:");
        topStudents.forEach(System.out::println);
    }}
```

OUTPUT:



```
Students scoring above 75%, sorted by marks:
Naksh
Akshita
Khushi


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

Learning Outcomes:

- Learn how to use lambda expressions for concise and readable code in Java.
- Understand how to apply filtering operations using the filter() method to extract relevant data.
- Explore how to transform objects using map() to extract specific attributes

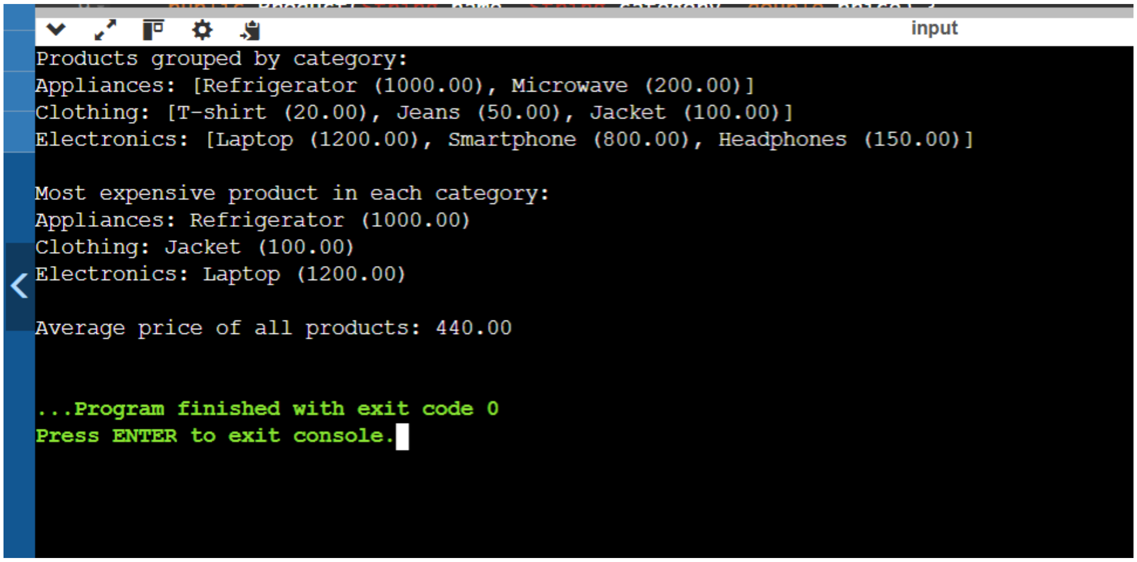
Problem 3: Write a Java program to process a large dataset of products using streams. Perform operations such as grouping products by category, finding the most expensive product in each category, and calculating the average price of all products**.**

Objective :

Code :

```
import java.util.*;

import java.util.stream.*;


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("%s (%.2f)", name, price);
    }
}

public class ProductProcessor {
    public static void main(String[] args) {
        List<Product> products = Arrays.asList(
            new Product("Laptop", "Electronics", 1200.00),
            new Product("Smartphone", "Electronics", 800.00),
            new Product("Headphones", "Electronics", 150.00),
            new Product("T-shirt", "Clothing", 20.00),
            new Product("Jeans", "Clothing", 50.00),
            new Product("Refrigerator", "Appliances", 1000.00),
            new Product("Microwave", "Appliances", 200.00),
            new Product("Jacket", "Clothing", 100.00)
        );
        Map<String, List<Product>> productsByCategory = products.stream()
            .collect(Collectors.groupingBy(Product::getCategory));

        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(
                Product::getCategory,
                Collectors.maxBy(Comparator.comparingDouble(Product::getPrice))
            ));
        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(Product::getPrice)
            .average()
            .orElse(0.0);

        System.out.printf("\nAverage price of all products: %.2f%n", averagePrice);
    }
}
```

OUTPUT:



```
Products grouped by category:
Appliances: [Refrigerator (1000.00), Microwave (200.00)]
Clothing: [T-shirt (20.00), Jeans (50.00), Jacket (100.00)]
Electronics: [Laptop (1200.00), Smartphone (800.00), Headphones (150.00)]

Most expensive product in each category:
Appliances: Refrigerator (1000.00)
Clothing: Jacket (100.00)
Electronics: Laptop (1200.00)

Average price of all products: 440.00


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

Learning Outcomes :

- Learn how to efficiently process large datasets using Java Streams.
- Learn how to use mapToDouble() and average() to compute the average price of all products.
- Understand how Function<T, R> and Comparator<T> are used in stream operations.