Experiment 6

Student Name: Rohit Soni
Branch: BE/CSE
Semester: 6th
Subject Name: Project Based Learning in JAVA with Lab

UID: 22BCS10110
Section/Group: 22BCS_IOT-618/A
Date of Performance: 28/02/25
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:** The objective of this practical is to implement Java programs using lambda expressions and stream operations for sorting, filtering, and processing large datasets efficiently.

3. **Implementaion\Code:**

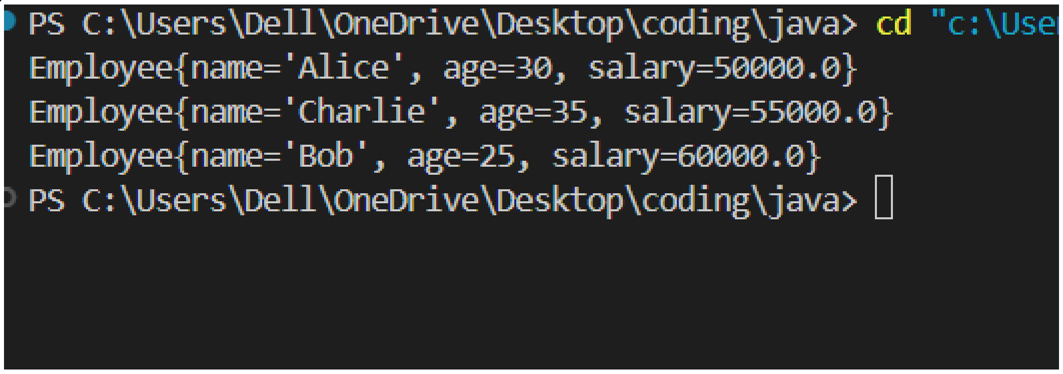
6.1 : Write a program to sort a list of Employee objects (name, age, salary) using lambda expressions.

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 = Arrays.asList(
            new Employee("Alice", 30, 50000),
            new Employee("Bob", 25, 60000),
            new Employee("Charlie", 35, 55000)
        );
        employees.sort((e1, e2) -> Double.compare(e1.salary, e2.salary));
        employees.forEach(System.out::println);
    }
}
```

Output:



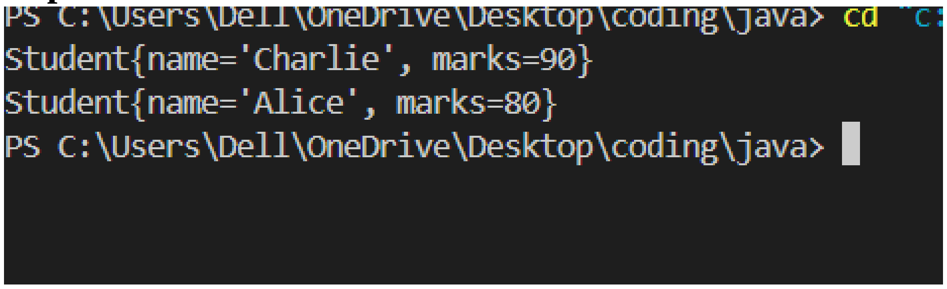
```
PS C:\Users\Dell\OneDrive\Desktop\coding\java> cd "c:\Use
Employee{name='Alice', age=30, salary=50000.0}
Employee{name='Charlie', age=35, salary=55000.0}
Employee{name='Bob', age=25, salary=60000.0}
PS C:\Users\Dell\OneDrive\Desktop\coding\java>
```

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

Code:

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

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

    @Override
    public String toString() {
        return "Student{name='" + name + "', marks=" + marks + "}";
    }
}

public class StudentFilterSort {
    public static void main(String[] args) {
        List<Student> students = Arrays.asList(
            new Student("Alice", 80),
            new Student("Bob", 65),
            new Student("Charlie", 90),
            new Student("David", 70)
        );
        List<Student> filteredSortedStudents = students.stream()
            .filter(s -> s.marks > 75)
            .sorted((s1, s2) -> Integer.compare(s2.marks, s1.marks))
            .collect(Collectors.toList());
        filteredSortedStudents.forEach(System.out::println);
    }
}
```

Output:



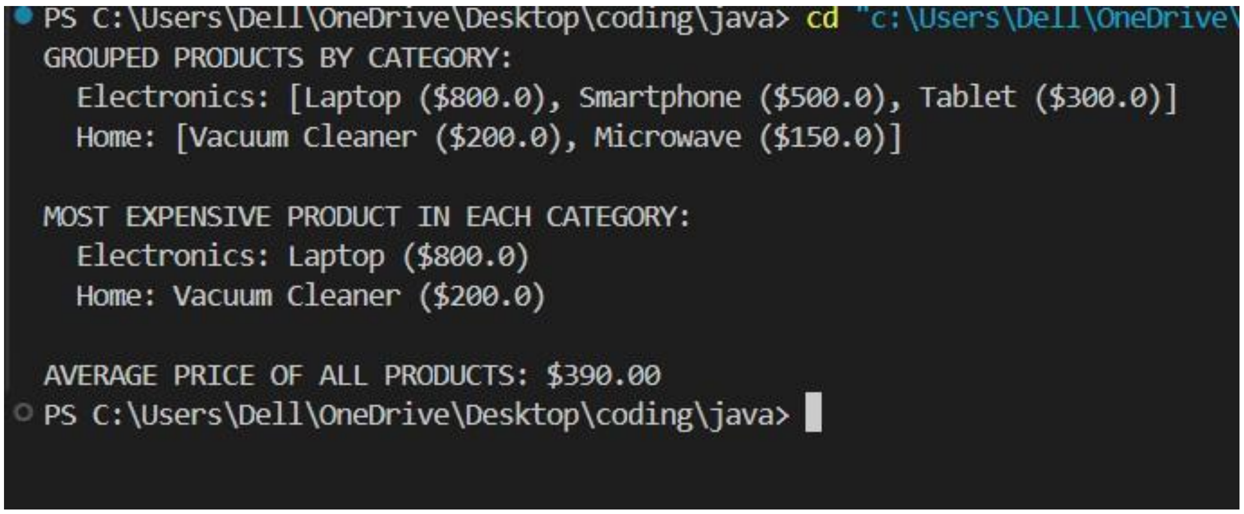
```
PS C:\Users\Dell\OneDrive\Desktop\coding\java> cd "c:
Student{name='Charlie', marks=90}
Student{name='Alice', marks=80}
PS C:\Users\Dell\OneDrive\Desktop\coding\java>
```

6.3: 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.

Code:

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

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

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

    @Override
    public String toString() {
        return name + " ($" + price + ")";
    }
}

public class ProductProcessing {
    public static void main(String[] args) {
        List<Product> products = Arrays.asList(
            new Product("Electronics", "Laptop", 800),
            new Product("Electronics", "Smartphone", 500),
            new Product("Home", "Vacuum Cleaner", 200),
            new Product("Home", "Microwave", 150),
            new Product("Electronics", "Tablet", 300)
        );
        Map<String, List<Product>> groupedByCategory = products.stream()
            .collect(Collectors.groupingBy(p -> p.category));
        Map<String, Product> mostExpensiveProduct = products.stream()
            .collect(Collectors.toMap(
                p -> p.category,
                p -> p,
                (p1, p2) -> p1.price > p2.price ? p1 : p2
            ));
        double avgPrice = products.stream()
            .mapToDouble(p -> p.price)
            .average()
            .orElse(0);
        System.out.println("GROUPED PRODUCTS BY CATEGORY:");
        groupedByCategory.forEach((category, productList) -> {
            System.out.println("  " + category + ": " + productList);
        });

        System.out.println("\nMOST EXPENSIVE PRODUCT IN EACH CATEGORY:");
        mostExpensiveProduct.forEach((category, product) -> {
            System.out.println("  " + category + ": " + product);
        });

        System.out.println("\nAVERAGE PRICE OF ALL PRODUCTS: $" + String.format("%.2f",
avgPrice));
    }
}
```

Output:



```
PS C:\Users\Dell\OneDrive\Desktop\coding\java> cd "c:\Users\Dell\OneDrive\
GROUPED PRODUCTS BY CATEGORY:
  Electronics: [Laptop ($800.0), Smartphone ($500.0), Tablet ($300.0)]
  Home: [Vacuum Cleaner ($200.0), Microwave ($150.0)]

MOST EXPENSIVE PRODUCT IN EACH CATEGORY:
  Electronics: Laptop ($800.0)
  Home: Vacuum Cleaner ($200.0)

AVERAGE PRICE OF ALL PRODUCTS: $390.00
PS C:\Users\Dell\OneDrive\Desktop\coding\java>
```

4. Learning Outcomes:

- Using Lambda Expressions & Streams – Learned how to apply lambda expressions for sorting, filtering, and processing data in Java.
- Database Connectivity with JDBC – Understood how to connect Java programs to MySQL databases and perform CRUD operations.
- MVC Architecture in Java – Implemented the Model-View-Controller (MVC) pattern for better separation of concerns in database applications.
- Data Processing & Aggregation – Used Java streams to group, filter, and compute statistics (like average price and max values) on datasets.
- Functional Programming Concepts – Practiced method references, functional interfaces, and stream operations for cleaner and more efficient code.