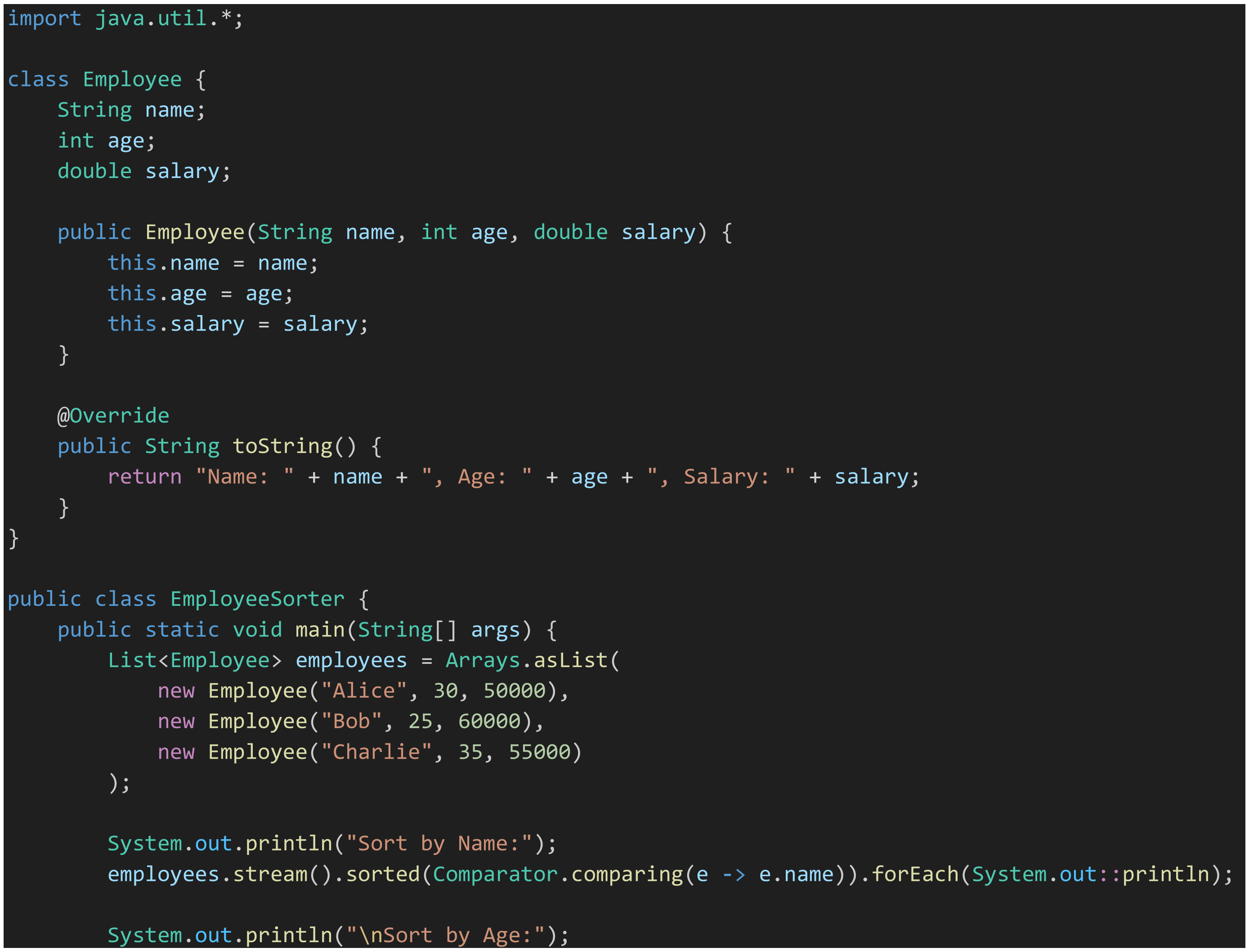
Experiment 6

Student Name: Vishal Sah **UID:22BCS16978**
Branch:BE/CSE **Section/Group: 641/B**
Semester: 6 **Date of Performance:06/03/25**
Subject Name:JAVA **Subject Code: 22CSH-359**

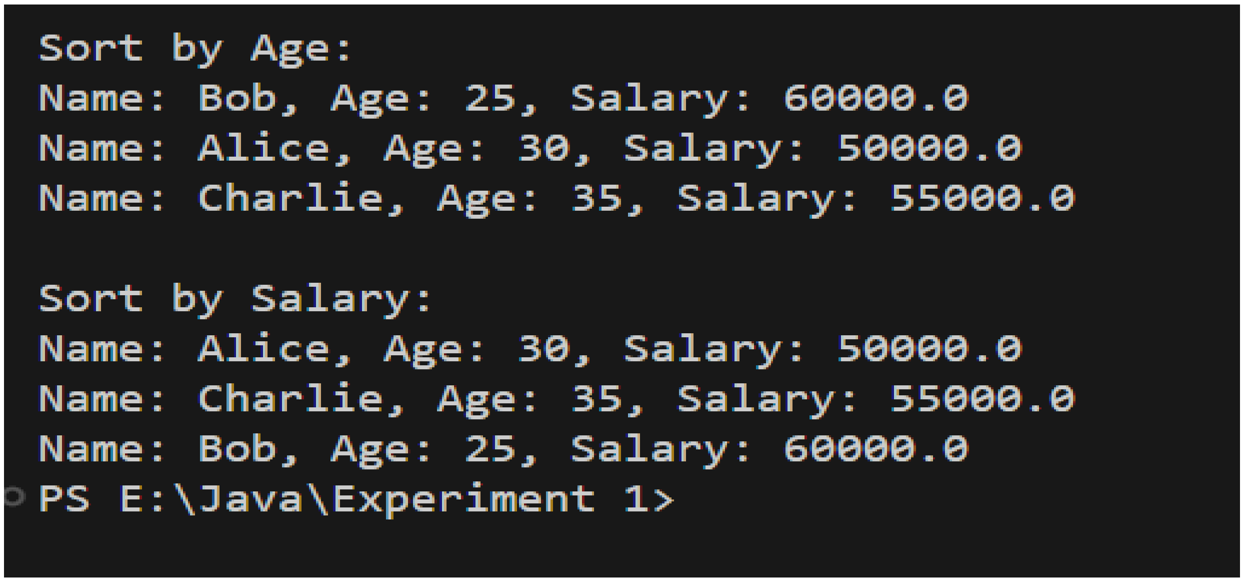
1. **Aim:** Write a program to sort a list of Employee objects (name, age, salary) using lambda expressions.

2. **Implementation/Code:**



```java
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 "Name: " + name + ", Age: " + age + ", Salary: " + salary;
    }
}

public class EmployeeSorter {
    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)
        );

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

        System.out.println("\nSort by Age:");
        employees.stream().sorted(Comparator.comparingInt(e ->
e.age)).forEach(System.out::println);

        System.out.println("\nSort by Salary:");
        employees.stream().sorted(Comparator.comparingDouble(e ->
e.salary)).forEach(System.out::println);
    }
}
```

3. Output



```
Sort by Age:
Name: Bob, Age: 25, Salary: 60000.0
Name: Alice, Age: 30, Salary: 50000.0
Name: Charlie, Age: 35, Salary: 55000.0

Sort by Salary:
Name: Alice, Age: 30, Salary: 50000.0
Name: Charlie, Age: 35, Salary: 55000.0
Name: Bob, Age: 25, Salary: 60000.0
PS E:\Java\Experiment 1>
```

4. Learning Outcome

1. **Lambda Expressions:** Understand how to use lambda expressions to create concise and functional code, especially for sorting with `Comparator`.

2. **Stream API Proficiency:** Gain experience using Java Streams for data processing, including sorting and displaying collections.

3. **Comparator Usage:** Learn to sort objects based on different fields (`name`, `age`, `salary`) using `Comparator.comparing`, `Comparator.comparingInt`, and `Comparator.comparingDouble`.

4. **Object-Oriented Programming (OOP) Practice**: Reinforce OOP principles by working with custom `Employee` objects and implementing the `toString()` method for easy output.

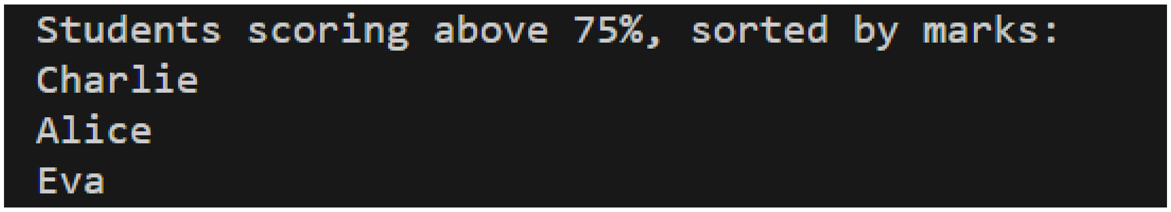
Experiment – 6.2

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

2- **Implementation/Code:**

```java
import java.util.*;

class Student {
    String name;
    double marks;

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

public class StudentFilterAndSort {
    public static void main(String[] args) {
        List<Student> students = Arrays.asList(
            new Student("Alice", 85.0),
            new Student("Bob", 72.5),
            new Student("Charlie", 78.3),
            new Student("David", 65.0),
            new Student("Eva", 91.0)
        );

        System.out.println("Students scoring above 75%, sorted by marks:");
        students.stream()
                .filter(s -> s.marks > 75)
                .sorted(Comparator.comparingDouble(s -> s.marks))
                .map(s -> s.name)
                .forEach(System.out::println);
    }
}
```

3- Output



```
Students scoring above 75%, sorted by marks:
Charlie
Alice
Eva
```

4- Learning Outcome :

1. **Lambda Expressions and Stream API:** Understand how to use lambda expressions with stream operations like `filter()`, `sorted()`, and `map()` to process collections efficiently.

2. **Data Filtering:** Learn to filter out specific data based on conditions (e.g., students scoring above 75%) using the `filter()` method.

3. **Sorting Collections:** Gain experience with `Comparator.comparingDouble()` to sort a list of custom objects by specific fields (marks in this case).

4. **Functional Programming in Java:** Develop skills in functional programming paradigms by combining multiple stream operations in a clean and readable manner.

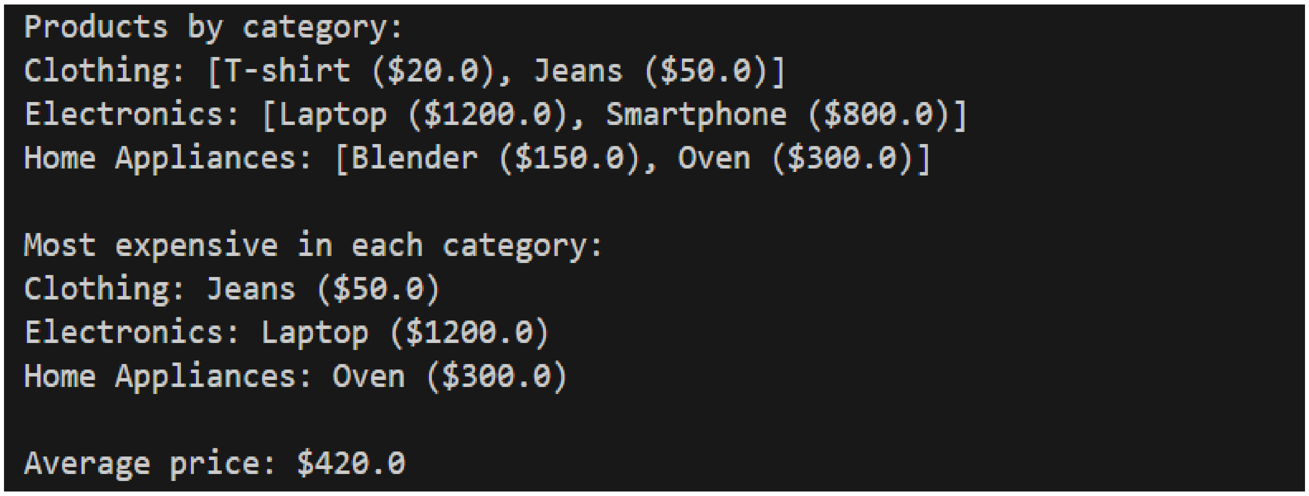
Experiment 6.3

1- **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.

2- **Implementation/Code:**

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

class Product {
    String name, 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", 1200.0),
            new Product("Smartphone", "Electronics", 800.0),
            new Product("T-shirt", "Clothing", 20.0),
            new Product("Jeans", "Clothing", 50.0),
            new Product("Blender", "Home Appliances", 150.0),
            new Product("Oven", "Home Appliances", 300.0)
        );

        System.out.println("Products by category:");
        products.stream()
                .collect(Collectors.groupingBy(p -> p.category))
                .forEach((cat, list) -> System.out.println(cat + ": " + list));

        System.out.println("\nMost expensive in each category:");
        products.stream()
                .collect(Collectors.groupingBy(p -> p.category, Collectors.maxBy(Comparator.comparingDouble(p
-> p.price))))
                .forEach((cat, prod) -> System.out.println(cat + ": " + prod.orElse(null)));
        double avgPrice = products.stream().mapToDouble(p -> p.price).average().orElse(0.0);
        System.out.println("\nAverage price: $" + avgPrice);
    }
}
```

3- Output



```
Products by category:
Clothing: [T-shirt ($20.0), Jeans ($50.0)]
Electronics: [Laptop ($1200.0), Smartphone ($800.0)]
Home Appliances: [Blender ($150.0), Oven ($300.0)]

Most expensive in each category:
Clothing: Jeans ($50.0)
Electronics: Laptop ($1200.0)
Home Appliances: Oven ($300.0)

Average price: $420.0
```

4- Learning Outcome :

1. **Stream API Mastery:** Learn how to effectively use Java Streams for processing collections, including grouping, filtering, and aggregation operations.

2. **Lambda Expressions:** Gain experience with lambda expressions to create concise and readable code, especially when used with `Collectors` and `Comparator`.

3. **Data Aggregation Techniques:** Understand how to group elements using `Collectors.groupingBy()` and find the maximum value in a group with `Collectors.maxBy()`.

4. **Statistical Calculation:** Develop the ability to calculate aggregate values such as the average price using `mapToDouble()` and `average()` methods in the Stream API.