Experiment:-6

Student Name: Harshad Fozdar
UID: 22BCS10263
Branch: CSE
Section/Group: 22BCS_DL-901/A
Semester: 6th
Date of Performance: 13/03/2025
Subject Name: PBLJ
Subject Code: 22CSH-359

Problem -1

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

2. **Objective:**

- **Create Employee Class** – Create an employee class with attributes like name, age, and salary to store employee details. This will help in organizing and managing employee data effectively.
- **Fix Empty Names** – If the employee's name is empty, assign "Unknown" to avoid blank values. This ensures that all employee names are properly displayed.
- **Sort Using Lambda** – Use lambda expressions to sort the list by name (alphabetically), age (ascending), and salary (ascending). This makes sorting simple and easy to understand.
- **Show Sorted Results** – Display the list of employees after each sorting operation. This helps to confirm that the sorting is working correctly.
- **Keep Code Simple** – Use lambda expressions to write clean and simple code. This improves code readability and makes it easy to modify if needed.

3. **Implementation/Code:**

```
import java.util.*;
 class Employee
  { String name;
   int age;
   double salary;
   public Employee(String name, int age, double salary)
      { this.name = name.isEmpty() ? "Unknown" : name;
      this.age = age;
      this.salary = salary;
   }
   @Override
   public String toString() {
      return String.format("%-10s | Age: %-3d | Salary: ₹%.2f", name, age, salary);
   }
}
public class EmployeeSort {
   public static void main(String[] args)
      { List<Employee> employees = new ArrayList<>();
      employees.add(new Employee("Surbhi", 20, 50000));
```

```
        employees.add(new Employee("Saumya", 21, 60000));
        employees.add(new Employee("Harsh", 19, 55000));
        System.out.println("\n=== Employee List ===");
        employees.forEach(System.out::println);  employees.sort((e1,
        e2) -> e1.name.compareTo(e2.name));
        System.out.println("\n=== Sorted by Name ===");
        employees.forEach(System.out::println);  employees.sort((e1,
        e2) -> Integer.compare(e1.age, e2.age));
        System.out.println("\n=== Sorted by Age ===");
        employees.forEach(System.out::println);
        employees.sort((e1, e2) -> Double.compare(e1.salary, e2.salary));
        System.out.println("\n=== Sorted by Salary ===");
        employees.forEach(System.out::println);
    }
}
```

4. Output:

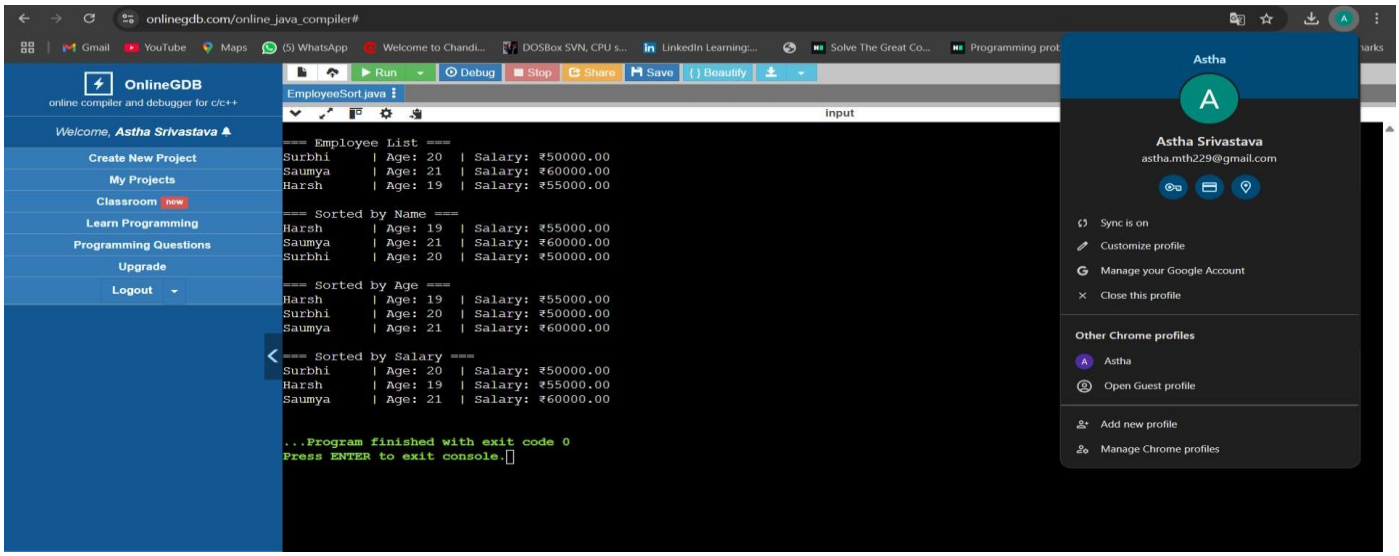

Figure 1

5. Learning Outcome:

- **Understanding Classes and Objects: -** Learn how to create and use classes and objects to store employee details.
- **Handling Empty Values: -** Understand how to handle empty names by setting a default value like "Unknown."
- **Using Lambda for Sorting: -** Learn how to use lambda expressions to sort data easily by name, age, and salary.
- **Displaying Results: -** Gain the skill to display sorted lists clearly to check if sorting works correctly.

Problem-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. **Objectives:**
   - **Create Student Class** – Create a student class with details like name and marks. This will help in storing and managing student data.
   - **Add Student Data** – Create a list of students with their names and marks. This makes it easy to apply filtering and sorting operations.
   - **Filter Top Students** – Use a lambda expression to filter students who scored more than 75%. This helps to focus only on high scorers.
   - **Sort by Marks** – Use lambda to sort the filtered list in descending order of marks. This shows the highest scorers at the top.
   - **Display Results** – Display the names of top students clearly. This helps to confirm that filtering and sorting work correctly.
3. **CODE:**

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
class Student {
   String name;
   double marks;
   public Student(String name, double marks)
      { this.name = name;
      this.marks = marks;
   }
}
public class Main {
   public static void main(String[] args)
      { List<Student> students = new ArrayList<>();
      students.add(new Student("Astha", 85));
      students.add(new Student("Khushi", 73));
      students.add(new Student("Ankit", 65));
      students.add(new Student("Kartik", 92));
      students.add(new Student("Anuska", 98));
      List<String> topStudents = students.stream()
         .filter(s -> s.marks > 75)
         .sorted((s1, s2) -> Double.compare(s2.marks, s1.marks))
         .map(s -> s.name)
         .collect(Collectors.toList());
      System.out.println("Students scoring above 75%:");
      topStudents.forEach(System.out::println);
```

```
    }
}
```

4. **Output:**

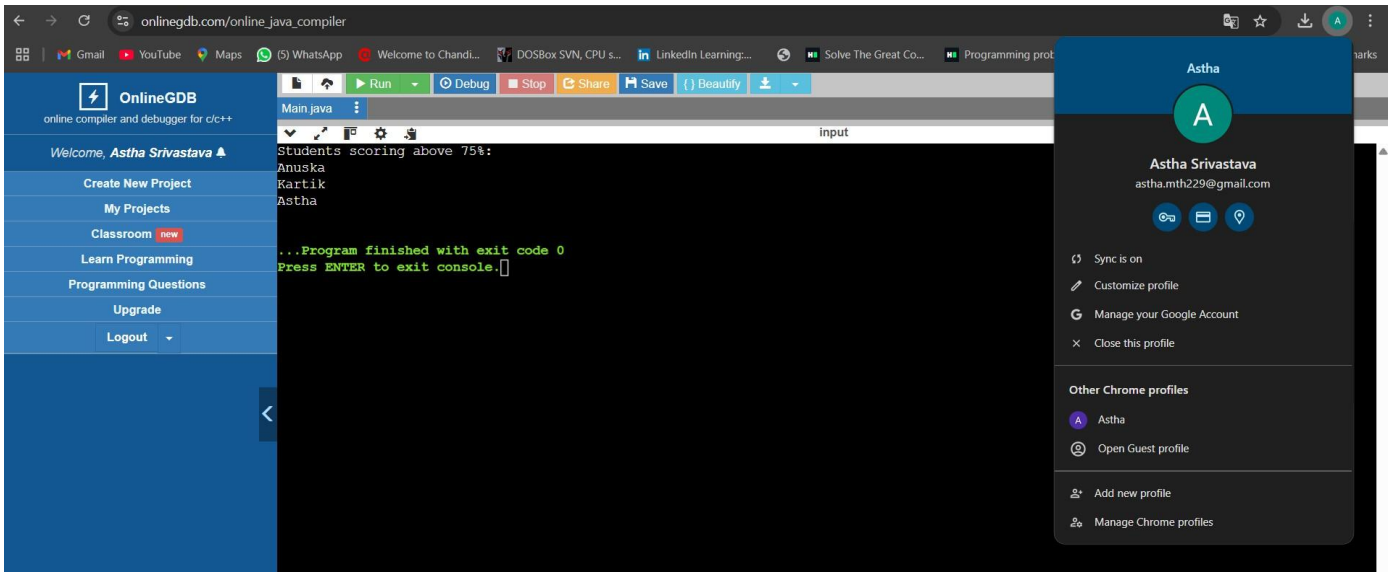

Figure 2

5. Learning Outcomes:

- **Understanding Classes and Objects: -** Learn how to create a Student class to store student names and marks. This helps in organizing and managing student data easily.
- **Using Stream and Lambda: -** Understand how to apply stream and lambda expressions to perform operations like filtering and sorting. This makes the code more efficient and readable.
- **Filtering Data: -** Learn how to filter students scoring above 75% using stream operations. This helps in selecting only the required data based on a condition.
- **Sorting with Lambda: -** Understand how to use lambda expressions to sort students by marks in descending order. This helps in displaying the highest scores first.
- **Displaying Results: -** Learn how to display the names of filtered and sorted students clearly. This confirms that the operations are working correctly.

Problem-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. Objectives:

- **Understanding Streams in Java:** Learn how to use Java streams to process large sets of data efficiently, making it easier to handle collections of objects like products.
- **Grouping Products by Category:** Understand how to group products into different categories using Java collections and streams, simplifying data organization.
- **Finding Most Expensive Product:** Learn to use streams to find the highest-priced product in each category, helping to analyze product pricing.
- **Calculating Average Price:** Learn to compute the average price of all products using streams, improving data analysis and calculation skills.
- **Enhancing Problem-Solving Skills:** Improve logical thinking and coding skills by implementing complex data processing tasks using Java.

3. Implementation/Code:

```
import java.util.*;
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 + " | Category: " + category + " | Price: ₹" + price;
   } }
public class ProductStream {
   public static void main(String[] args)
      { List<Product> products = Arrays.asList(
         new Product("Laptop", "Electronics", 70000),
         new Product("Mobile", "Electronics", 30000),
         new Product("Shirt", "Clothing", 1500),
         new Product("Jeans", "Clothing", 2500),
         new Product("Mixer", "Home Appliance", 4000),
         new Product("TV", "Electronics", 60000) );
      System.out.println("\n=== Product List ===");
      for (Product p : products) {
         System.out.println(p);}
```

```
        System.out.println("\n=== Grouped by Category ===");
        Map<String, List<Product>> grouped = new HashMap<>();
        for (Product p : products) {
            grouped.computeIfAbsent(p.category, k -> new ArrayList<>()).add(p); }
        for (String category : grouped.keySet()) {
            System.out.println("\n" + category + ":");
            for (Product p : grouped.get(category)) {
                System.out.println(p);
            } }
        System.out.println("\n=== Most Expensive Product in Each Category ===");
        for (String category : grouped.keySet()) {
            Product maxProduct = Collections.max(grouped.get(category),
    Comparator.comparingDouble(p -> p.price));
            System.out.println(category + ": " + maxProduct); }
        double total = 0;
        for (Product p : products)
            { total += p.price; }
        double average = total / products.size();
        System.out.printf("\n=== Average Price of All Products ===\n₹%.2f\n", average);
    } }
```

4. Output:

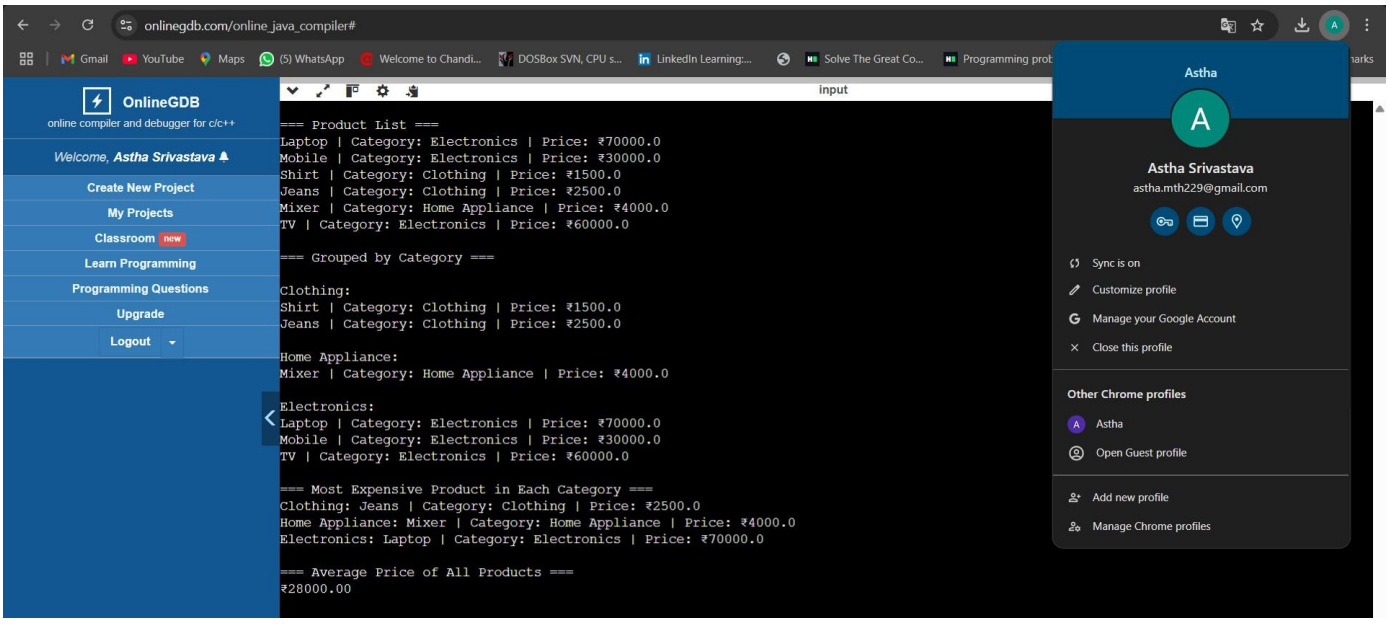

Figure 2

5. Learning Outcomes:

- **Stream Operations:** Students will be able to use Java streams to filter, map, and process large sets of data efficiently.
- **Data Grouping:** Students will know how to group products based on their categories using Java collections and streams.
- **Identifying High-Value Items:** Students will be able to find the most expensive product in each category using Java streams and comparators.
- **Average Calculation:** Students will be able to calculate the average value of data elements using streams and basic math operations.
- **Code Optimization:** Students will be able to write cleaner and more efficient code by applying stream-based data processing techniques.