Experiment 6

Student Name: Piyush Raj **UID: 22BCS14113**
Branch: CSE **Section: 22BCS_DL-903'B**
Semester: 6th **DOP:21/02/25**
Subject: PBLJ **Subject Code:22CSH-359**

Easy

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

Objective: Develop Java programs using lambda expressions and stream operations for sorting large datasets efficiently.

Algorithm:

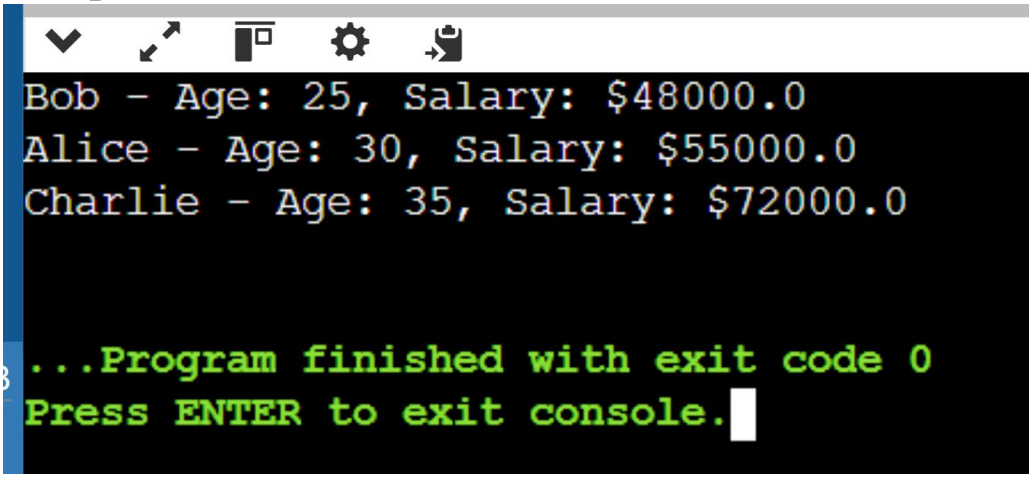
1. Define an Employee class with attributes name, age, and salary.
2. Create a list of Employee objects with predefined values.
3. Sort the list using a lambda expression with Comparator.comparingDouble(emp -> emp.salary).
4. Display the sorted list using forEach().

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;
    }
    @Override
    public String toString() {
        return name + " - Marks: " + marks;
    }
}
```

```
public class StudentProcessing {
    public static void main(String[] args) {
        List<Student> students = Arrays.asList(
            new Student("Alice", 82),
            new Student("Bob", 67),
            new Student("Charlie", 90),
            new Student("David", 74),
            new Student("Eve", 88)
        );
        List<String> topStudents = students.stream()
                .filter(s -> s.marks > 75)
                .sorted(Comparator.comparingDouble(s -> -s.marks))
                .map(s -> s.name)
                .collect(Collectors.toList());
        topStudents.forEach(System.out::println);
    }
}
```

Output:



Learning Outcomes:

1. **Understanding Lambda Expressions** – Learn how to use lambda expressions to define sorting logic concisely.
2. **Comparator Functional Interface** – Use Comparator.comparing() to simplify object comparison.
3. **List Sorting Techniques** – Gain insights into sorting objects using List.sort() and lambda expressions.
4. **Method References** – Understand how System.out::println simplifies printing collections.
5. **Immutable vs. Mutable Lists** – Recognize how sorting modifies a mutable list in-place.

Medium

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.Objective: Develop Java programs using lambda expressions and stream operations for filtering large datasets efficiently.

3.Algorithm:

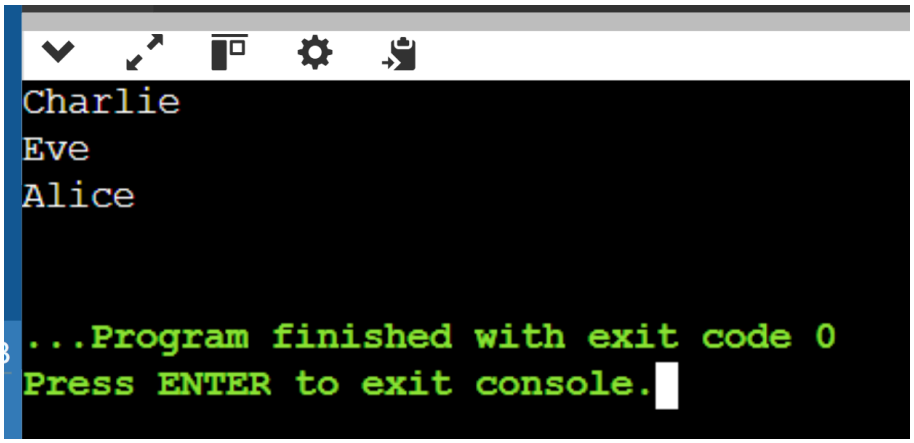
- Define a Student class with attributes name and marks.
- Create a list of Student objects with sample data.
- Filter students scoring above 75% using filter().
- Sort the filtered students in descending order of marks using sorted().
- Extract only the student names using map().
- Collect the results into a list using collect().
- Print the list of filtered and sorted student names.

4. Implementation 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;
   }

   @Override
   public String toString() {
      return name + " - Marks: " +
marks;
   }
}

public class StudentProcessing {
   public static void main(String[]
args) {
```

```
        List<Student> students =
Arrays.asList(
            new Student("Alice", 82),
            new Student("Bob", 67),
            new Student("Charlie", 90),
            new Student("David", 74),
            new Student("Eve", 88)
        );

        // Filter students scoring above
75%, sort by marks, and display
names
        List<String> topStudents =
students.stream()
                .filter(s -> s.marks > 75)

.sorted(Comparator.comparingDouble(
s -> -s.marks))
                .map(s -> s.name)
                .collect(Collectors.toList());

        // Display result

topStudents.forEach(System.out::print
ln);
    }
}}
```

5.Output



6.Learning Outcomes:

- **Stream API Basics** – Learn how to filter, sort, and transform collections efficiently.
- **Functional Programming Concepts** – Understand the benefits of declarative programming using streams.

- **Filtering Data Efficiently** – Use filter() to extract specific elements based on conditions.
- **Sorting with Streams** – Apply sorted() with a comparator to order filtered data.
- **Mapping and Collecting Data** – Convert objects to another form (map()) and store results (collect()).

Hard

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. **Objective:** Develop Java programs using lambda expressions and stream operations for processing large datasets efficiently.

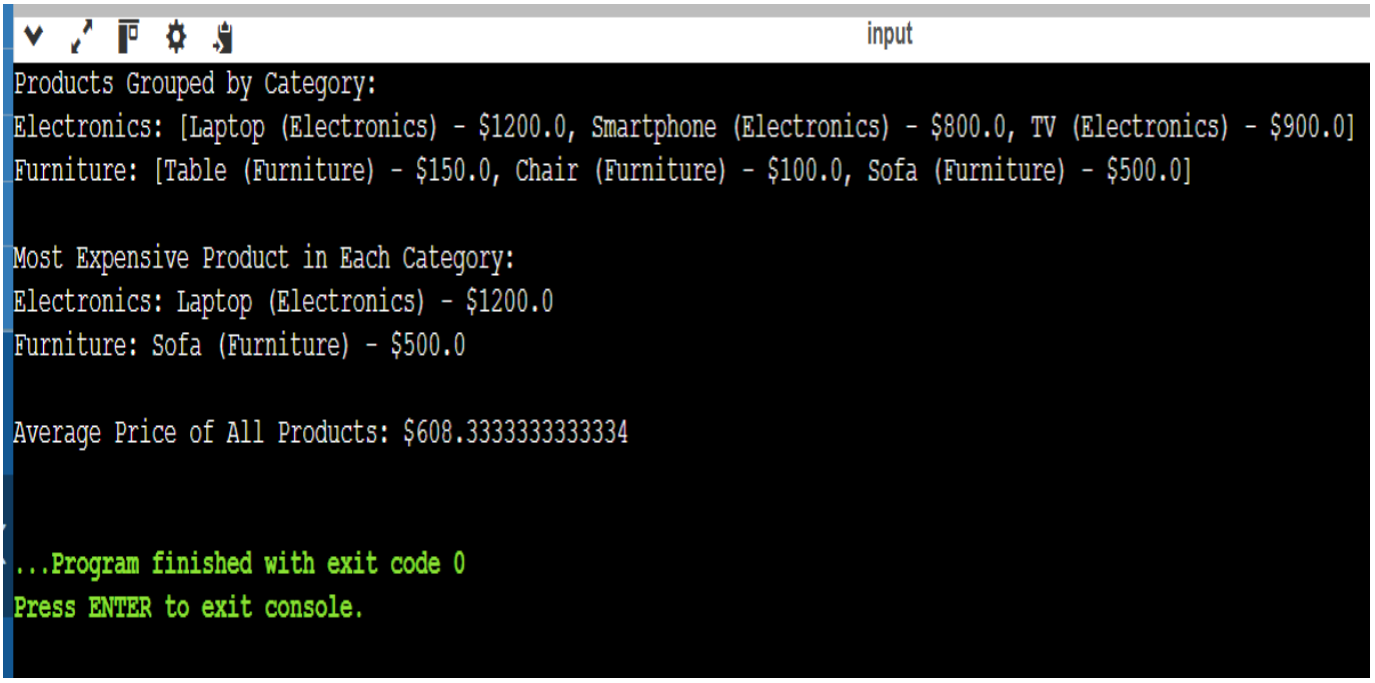
3. **Algorithm:**
   - Define a Product class with attributes name, category, and price.
   - Create a list of Product objects representing a dataset.
   - Group products by category using collect(Collectors.groupingBy()).
   - Find the most expensive product in each category using collect(Collectors.maxBy(Comparator.comparingDouble())).
   - Calculate the average price of all products using mapToDouble().average().
   - Display the grouped products.
   - Display the most expensive product in each category.
   - Display the average price of all products.

4.Implementation Code:

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

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 + ") - $" + price;
    }
}
```

```
public class ProductProcessing {
    public static void main(String[] args) {
        List<Product> products = Arrays.asList(
            new Product("Laptop", "Electronics", 1200),
            new Product("Smartphone", "Electronics", 800),
            new Product("Table", "Furniture", 150),
            new Product("Chair", "Furniture", 100),
            new Product("TV", "Electronics", 900),
            new Product("Sofa", "Furniture", 500)
        );

        Map<String, List<Product>> groupedByCategory = products.stream()
                .collect(Collectors.groupingBy(p -> p.category));

        Map<String, Optional<Product>> mostExpensiveByCategory = products.stream()
                .collect(Collectors.groupingBy(p -> p.category,
                        Collectors.maxBy(Comparator.comparingDouble(p -> p.price))));

        double averagePrice = products.stream()
                .mapToDouble(p -> p.price)
                .average()
                .orElse(0);

        System.out.println("Products Grouped by Category:");
        groupedByCategory.forEach((category, list) -> {
            System.out.println(category + ": " + list);
        });

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

        System.out.println("\nAverage Price of All Products: $" + averagePrice);
    }
}
```

5. Output:



6. Learning Outcomes:

- **Advanced Stream Operations** – Learn about groupingBy(), maxBy(), and mapToDouble() for complex data processing.
- **Data Aggregation Techniques** – Perform operations like categorization, finding maximum values, and averaging efficiently.
- **Using Collectors for Grouping** – Master Collectors.groupingBy() to structure and organize large datasets.
- **Efficient Data Processing** – Optimize handling of large datasets without manual iteration using streams.
- **Combining Multiple Operations** – Learn how to chain multiple operations (filtering, grouping, and reducing) in a single stream pipeline.