

## Experiment 6.1

**Student Name:** Anukriti

**Branch:** CSE

**Semester:** 6<sup>th</sup>

**Subject:** PBLJ

**UID:**22BCS11150.

**Section:** EPAM-801(B)

**DOP:**24.02.25

**Subject Code:**22CSH-359

**Aim:** Develop Java programs using lambda expressions and stream operations for sorting, filtering, and processing large datasets efficiently. a.)Write a program to sort a list of Employee objects (name, age, salary) using lambda expressions.

**Objective:** To sort a list of Employee objects based on different attributes (name, age, and salary) using lambda expressions in Java.

### **Algorithm:**

#### **1. Initialize Employee Data:**

- Define an Employee class with attributes: name, age, and salary.
- Implement a toString() method for formatted output.
- Create a list of Employee objects with sample data.

#### **2. Display Sorting Options:**

- Use a while (true) loop to continuously prompt the user for sorting options.
- Display a menu:
  - 1 → Sort by Name.
  - 2 → Sort by Age.
  - 3 → Sort by Salary (with additional filtering options).
  - 4 → Exit the program.

#### **3. User Choice Handling:**

- Read the user's choice using Scanner.
- Create a copy of the original employees list to perform sorting operations.

#### **4. Sorting Logic:**

- Option 1: Sort by Name
  - Use Stream.sorted(Comparator.comparing(Employee::name)).
  - Display the sorted list.
- Option 2: Sort by Age
  - Use Stream.sorted(Comparator.comparingInt(Employee::age)).
  - Display the sorted list.
- Option 3: Sort by Salary
  - Prompt the user to select sorting order:
    - 1 → Ascending order.
    - 2 → Descending order.
  - Use Stream.sorted(Comparator.comparingDouble(Employee::salary)) for ascending.
  - Use Stream.sorted(Comparator.comparingDouble(e -> -e.salary)) for descending.

#### **5. Filtering by Salary (Only for Salary Sorting):**

- Ask the user to input a salary threshold.
- Ask whether to display employees:
  - 1 → Above or equal to the threshold.
  - 2 → Below or equal to the threshold.
- Apply Stream.filter() to retain only the employees matching the condition.

#### **6. Display the Sorted & Filtered Employees.**

#### **7. Exit Option:**

- If the user selects option 4, display "Exiting...", close the scanner, and terminate the program.

## 8. Handle Invalid Input:

- If the user enters an invalid choice, display an error message and prompt again.

### Code:

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

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 String toString() {
        return String.format("Employee{ name='%s', age=%d, salary=%.2f}", name, age, salary);
    }
}

public class EmployeeSorting {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        List<Employee> employees = Arrays.asList(
            new Employee("Anu", 20, 50000),
            new Employee("Bhanu", 45, 6000),
            new Employee("Charu", 38, 14000),
            new Employee("Eva", 40, 78000),
            new Employee("Ishmeet", 18, 85000)
        );

        while (true) {
            System.out.println("\nChoose Sorting Option:");
            System.out.println("1. Sort by Name");
            System.out.println("2. Sort by Age");
            System.out.println("3. Sort by Salary");
            System.out.println("4. Exit");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();

            List<Employee> sortedList = new ArrayList<>(employees);

            switch (choice) {
                case 1:
                    sortedList = employees.stream()
                        .sorted(Comparator.comparing(e -> e.name))
                        .collect(Collectors.toList());
                    System.out.println("\nSorted by Name:");
                    break;
```

case 2:

```
sortedList = employees.stream()
    .sorted(Comparator.comparingInt(e -> e.age))
    .collect(Collectors.toList());
System.out.println("\nSorted by Age:");
break;
```

case 3:

```
System.out.print("Choose Salary Sorting Order (1 for Ascending, 2 for Descending): ");
int order = scanner.nextInt();
```

```
sortedList = employees.stream()
    .sorted(order == 1
        ? Comparator.comparingDouble(e -> e.salary)
        : Comparator.comparingDouble(e -> -e.salary))
    .collect(Collectors.toList());
```

```
System.out.println(order == 1 ? "\nSorted by Salary (Ascending):" : "\nSorted by Salary (Descending):");
```

```
// Asking for filter criteria
System.out.print("Enter the salary threshold: ");
double threshold = scanner.nextDouble();
```

```
System.out.print("Do you want employees (1) Above or (2) Below this salary? ");
int filterChoice = scanner.nextInt();
```

```
sortedList = sortedList.stream()
    .filter(e -> (filterChoice == 1 ? e.salary >= threshold : e.salary <= threshold))
    .collect(Collectors.toList());
```

```
System.out.println(filterChoice == 1 ? "\nEmployees with salary above or equal to " + threshold
+ ":"
: "\nEmployees with salary below or equal to " + threshold + ":" );
break;
```

case 4:

```
System.out.println("Exiting...");
scanner.close();
return;
```

default:

```
System.out.println("Invalid choice. Please try again.");
continue;
```

```
}
```

```
sortedList.forEach(System.out::println);
```

```
}
```

```
}
```

```
}
```

## Output:

```
Choose Sorting Option:
1. Sort by Name
2. Sort by Age
3. Sort by Salary
4. Exit
Enter your choice: 3
Choose Salary Sorting Order (1 for Ascending, 2 for Descending): 1

Sorted by Salary (Ascending):
Enter the salary threshold: 5000
Do you want employees (1) Above or (2) Below this salary? 1

Employees with salary above or equal to 5000.0:
Employee{name='Bhanu', age=45, salary=6000.00}
Employee{name='Charu', age=38, salary=14000.00}
Employee{name='Anu', age=20, salary=50000.00}
Employee{name='Eva', age=40, salary=78000.00}
Employee{name='Ishmeet', age=18, salary=85000.00}

Choose Sorting Option:
1. Sort by Name
2. Sort by Age
3. Sort by Salary
4. Exit
Enter your choice: 
```

## Learning Outcomes:

- **Efficient Data Processing with Java Streams & Lambda Expressions** – Learn how to use **lambda expressions** and the **Stream API** for sorting, filtering, and processing collections concisely.
- **Dynamic Sorting & Filtering Mechanisms** – Implement **Comparator.comparing()** for sorting by name, age, and salary, along with **filter()** to refine data based on user-defined conditions.
- **Interactive User Input Handling** – Develop **menu-driven programs** using Scanner to allow real-time user choices for sorting and filtering.
- **Working with Collections & Stream Operations** – Understand **ArrayLists**, **Collectors.toList()**, and **Stream pipelines** to efficiently manipulate and store processed data.
- **Robust and Scalable Program Design** – Ensure **error handling**, **input validation**, and **modular structure**, making the program scalable and user-friendly.

## Experiment 6.2

1. **Aim:** Develop Java programs using lambda expressions and stream operations for sorting, filtering, and processing large datasets efficiently. b.) 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:** To filter students who scored above **75%**, sort them in **descending order of marks**, and display their names using **Java Streams and Lambda Expressions**.

### 3. Algorithm:

#### Step 1: Define the Student Class

- Create a class Student with attributes:
  - name (String) → Stores the student's name.
  - marks (double) → Stores the student's marks.
- Implement the toString() method to **format output** when printing student objects.

#### Step 2: Initialize a List of Students

- Create a List<Student> containing predefined student objects with **name and marks**.

#### Step 3: Filter Students Who Scored Above 75%

- Use **Streams** to filter students with marks > 75 using:

#### Step 4: Sort the Filtered Students in Descending Order of Marks

- Use the sorted() method with a **custom comparator**:
- This sorts the remaining students (**Charu - 90, Emu - 88, Anu - 85**) in descending order.

#### Step 5: Extract Only Student Names

- Use .map(s -> s.name) to **convert Student objects to names**.

#### Step 6: Collect the Processed Data into a List

- Store the final names list using collect(Collectors.toList()).

#### Step 7: Print the Result

- Display the names of students **sorted in descending order of marks**.

## Implementation Code:

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

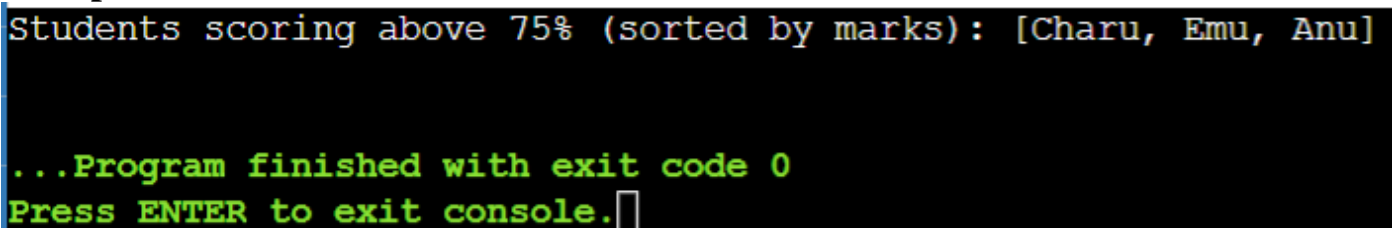
class Student {
    String name;
    double marks;

    public Student(String name, double marks) {
        this.name = name;
        this.marks = marks;
    }
    public String toString() {
        return name + " - Marks: " + marks;
    }
}

public class StudentFilterSort {
    public static void main(String[] args) {
        List<Student> students = Arrays.asList(
            new Student("Anu", 85),
            new Student("Bhanu", 72),
            new Student("Charu", 90),
            new Student("Donkey", 65),
            new Student("Emu", 88)
        );
        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% (sorted by marks): " + topStudents);
    }
}
```

## 4. Output



```
Students scoring above 75% (sorted by marks): [Charu, Emu, Anu]

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

## 5. Learning Outcomes:

- **Lambda Expressions & Stream API Usage** – Learn how to **filter, sort, and transform data** using Java Streams efficiently.
- **Sorting with Comparator & Stream Operations** – Understand **custom sorting logic** using `sorted()` with lambda expressions.
- **Filtering Data Dynamically** – Use `filter()` to **apply conditional logic** for data selection.

## Experiment 6.3

1. **Aim:** Develop Java programs using lambda expressions and stream operations for sorting, filtering, and processing large datasets efficiently. c.) 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:** To efficiently process a **large dataset of products** using **Java Streams** by performing the following operations:
  - **Grouping products** by category.
  - **Finding the most expensive product** in each category.
  - **Calculating the average price** of all products.

### 3. Algorithm:

#### **Step 1: Define the Product Class**

- Create a Product class with attributes:
  - name (String) → Name of the product.
  - category (String) → Category of the product.
  - price (double) → Price of the product.
- Implement toString() for formatted output.

#### **Step 2: Create a List of Products**

- Initialize a List<Product> with various product entries spanning multiple categories.

#### **Step 3: Group Products by Category**

- Use **Collectors.groupingBy()** to **group products** by their category.

#### **Step 4: Find the Most Expensive Product in Each Category**

- Use **Collectors.maxBy(Comparator.comparingDouble(Product::getPrice))** to determine the **most expensive product per category**.

#### **Step 5: Calculate the Average Price of All Products**

- Use **mapToDouble(Product::getPrice).average()** to compute the **average price of all products**.

#### **Step 6: Print the Results**

- Display the grouped products, the most expensive product in each category, and the average price.

### 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;
    }

    public double getPrice() {
        return price;
    }

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



```
}  
}  
  
public class ProductStreamProcessing {  
    public static void main(String[] args) {  
        List<Product> products = Arrays.asList(  
            new Product("Laptop", "Electronics", 800),  
            new Product("Smartphone", "Electronics", 600),  
            new Product("Headphones", "Electronics", 150),  
            new Product("Table", "Furniture", 200),  
            new Product("Chair", "Furniture", 100),  
            new Product("Sofa", "Furniture", 1200),  
            new Product("Rice", "Grocery", 50),  
            new Product("Milk", "Grocery", 30),  
            new Product("Apples", "Grocery", 40)  
        );  
        Map<String, List<Product>> groupedByCategory = products.stream()  
            .collect(Collectors.groupingBy(p -> p.category));  
        Map<String, Optional<Product>> mostExpensiveInCategory = products.stream()  
            .collect(Collectors.groupingBy(  
                p -> p.category,  
                Collectors.maxBy(Comparator.comparingDouble(Product::getPrice))  
            ));  
        double averagePrice = products.stream()  
            .mapToDouble(Product::getPrice)  
            .average()  
            .orElse(0.0);  
        System.out.println("Products Grouped by Category:");  
        groupedByCategory.forEach((category, productList) ->  
            System.out.println(category + ": " + productList));  
  
        System.out.println("\nMost Expensive Product in Each Category:");  
        mostExpensiveInCategory.forEach((category, product) ->  
            System.out.println(category + ": " + product.orElse(null)));  
  
        System.out.println("\nAverage Price of All Products: $" + String.format("%.2f",  
            averagePrice));  
    }  
}  
  
        System.out.println("\nAverage Price of All Products: $" + String.format("%.2f",  
            averagePrice));  
    }  
}  
  
System.out.println("Invalid  
choice! Please try again.");
```



## 5. Output:

```
Products Grouped by Category:
Grocery: [Rice ($50.0), Milk ($30.0), Apples ($40.0)]
Electronics: [Laptop ($800.0), Smartphone ($600.0), Headphones ($150.0)]
Furniture: [Table ($200.0), Chair ($100.0), Sofa ($1200.0)]

Most Expensive Product in Each Category:
Grocery: Rice ($50.0)
Electronics: Laptop ($800.0)
Furniture: Sofa ($1200.0)

Average Price of All Products: $352.22

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

## 6. Learning Outcomes:

- **Mastering Java Streams for Data Processing** – Learn how to **group, filter, sort, and aggregate** large datasets using **functional programming** in Java.
- **Efficient Data Grouping & Aggregation** – Use `Collectors.groupingBy()` to organize data based on categories.
- **Applying Comparators for Data Analysis** – Use `Collectors.maxBy(Comparator.comparingDouble())` to find the most expensive product in each category.
- **Computing Aggregate Statistics** – Utilize `mapToDouble().average()` to calculate the **average price** of products.