Experiment 6.1

Student Name: Vivek Garg	**UID: 22BCS16972**
Branch: CSE	**Section: 22BCS_EPAM-801/B**
Semester: 6th	**DOP:17/03/25**
Subject: PBLJ	**Subject Code:22CSH-359**

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

2. Objective:
Demonstrate sorting of Employee objects using **lambda expressions** in Java by sorting based on name, age, and salary while utilizing Comparator.comparing().

3. Algorithm:

Step 1: Initialize the Program

1. Start the program.
2. Import ArrayList, List and Comparator classes.
3. Define the Employee class with attributes.
    - Name
    - Age
    - Salary
4. Implement a constructor to initialize these attributes.
5. Override the toString() method for formatted output.

Step 2: Create a List of Employees

1. Define the main() method in the easyEmployeeSorter class.
2. Create a List<Employee> object using ArrayList<>.
3. Add Employee objects with sample data.

Step 3: Sort the Employee List

1. Sort by Name:
    - Use employees.sort(Comparator.comparing(emp -> emp.name));.
2. Sort by Age:
    - Use employees.sort(Comparator.comparingInt(emp -> emp.age));.
3. Sort by Salary:
    - Use employees.sort(Comparator.comparingDouble(emp -> emp.salary));.

Step 4: Print the Sorted List

1. Define a method printEmployees(List<Employee> employees).
2. Use a for loop to iterate through the list and print each Employee object.
3. Call printEmployees() after each sorting operation.

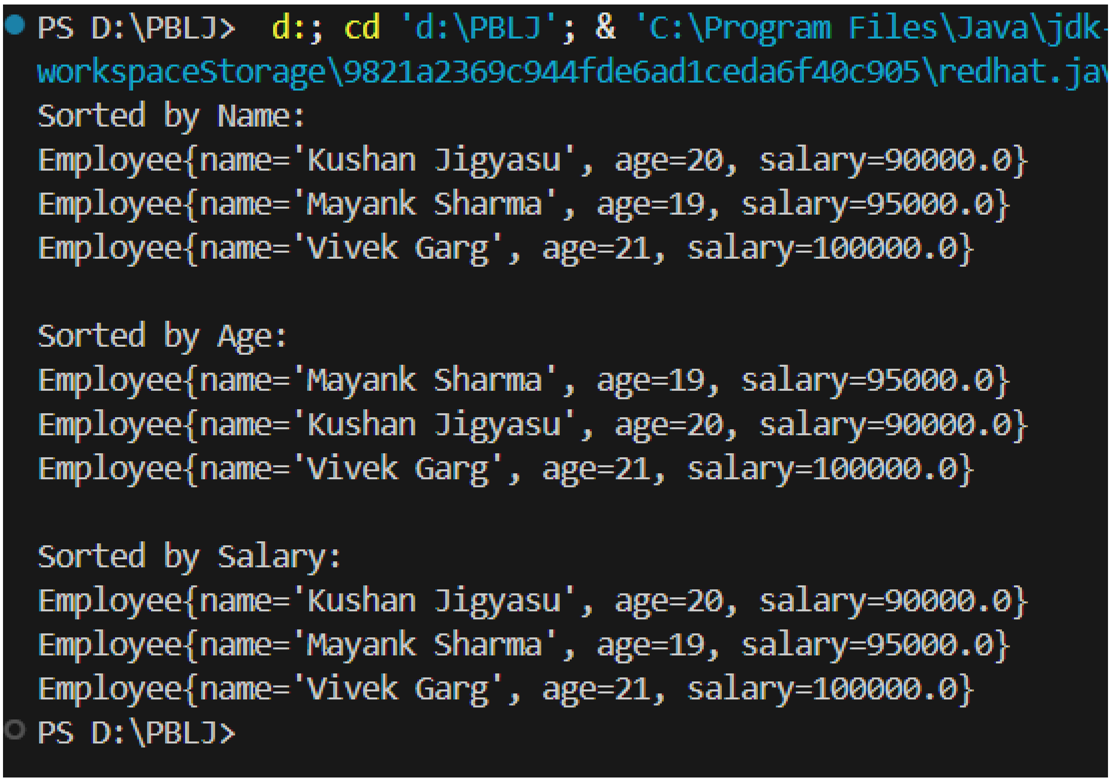
Step 5: Execute and Terminate the Program

1. Run the program.
2. Observe employees sorted by name, age and salary.
3. End the execution.

4. **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 easyEmployeeSorter {
    public static void main(String[] args) {
        List<Employee> employees = new ArrayList<>();
        employees.add(new Employee("Vivek Garg", 21, 60000));
        employees.add(new Employee("Mayank Sharma", 20, 55000));
        employees.add(new Employee("Kushan Jigyasu", 21, 50000));

        employees.sort(Comparator.comparing(emp -> emp.name));
        System.out.println("Sorted by Name:");
        printEmployees(employees);

        employees.sort(Comparator.comparingInt(emp -> emp.age));
        System.out.println("\nSorted by Age:");
        printEmployees(employees);

        employees.sort(Comparator.comparingDouble(emp -> emp.salary));
        System.out.println("\nSorted by Salary:");
        printEmployees(employees);
    }

    private static void printEmployees(List<Employee> employees) {
        for (Employee emp : employees) {
            System.out.println(emp);
        }
    }
}
```

5. **Output**:



```
PS D:\PBLJ>  d:; cd 'd:\PBLJ'; & 'C:\Program Files\Java\jdk-
workspaceStorage\9821a2369c944fde6ad1ceda6f40c905\redhat.ja
Sorted by Name:
Employee{name='Kushan Jigyasu', age=20, salary=90000.0}
Employee{name='Mayank Sharma', age=19, salary=95000.0}
Employee{name='Vivek Garg', age=21, salary=100000.0}

Sorted by Age:
Employee{name='Mayank Sharma', age=19, salary=95000.0}
Employee{name='Kushan Jigyasu', age=20, salary=90000.0}
Employee{name='Vivek Garg', age=21, salary=100000.0}

Sorted by Salary:
Employee{name='Kushan Jigyasu', age=20, salary=90000.0}
Employee{name='Mayank Sharma', age=19, salary=95000.0}
Employee{name='Vivek Garg', age=21, salary=100000.0}
PS D:\PBLJ>
```

6. **Learning Outcomes:**

- Understand the **concept of lambda expressions and functional programming** in Java to simplify sorting operations.
- Learn how to use the **Comparator.comparing() method** to sort objects based on different attributes dynamically.
- Gain experience in **working with ArrayList and List interfaces** to store and manipulate a collection of objects.
- Develop proficiency in using Java’s **sort() method** with custom comparators to sort by name, age, and salary efficiently.
- Enhance skills in iterating through collections and printing data using traditional for loops for structured output.
- Understand the **time complexity of sorting algorithms (O(n log n))** and how Java optimizes sorting operations internally.

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. **Objective:** Demonstrate the use of **lambda expressions & stream** in Java to filter students scoring above 75%.

3. **Algorithm:**

Step 1: Initialize the Program

1. Start the program.
2. Import ArrayList, List, Stream and Comparator classes.
3. Define the student class with attributes.
    - Name
    - Marks
4. Implement a constructor to initialize these attributes.
5. Override the toString() method for formatted output.

Step 2: Create a List of Students

1. Define the main() method in the mediumStudentFilter class.
2. Create a List<Student> using ArrayList<>.
3. Add multiple Student objects with sample data.

Step 3: Apply Stream Operations

1. Use **.stream()** to process the list.
2. Filter students scoring above 75% using .**filter(student -> student.marks > 75).**
3. Sort the filtered students by marks in descending order using **.sorted(Comparator.comparingDouble(Student::getMarks).reversed()).**
4. Extract only student names using **.map(Student::getName).**
5. Collect results into a list using **.toList().**

Step 4: Display the Filtered and Sorted Names

1. Iterate through the filtered list using a for loop and print names.

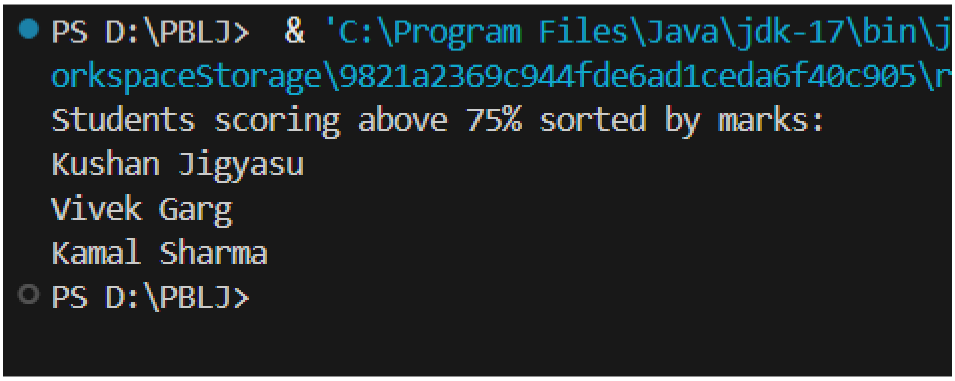
Step 5: Execute and Terminate the Program

1. Run the program.
2. Observe the filtered and sorted student names.
3. End the execution.

4. Code:

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

class Student {
    private String name;
    private double marks;

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

    public String getName() {
        return name;
    }

    public double getMarks() {
        return marks;
    }
}

public class mediumStudentFilter {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student("Vivek Garg", 85));
        students.add(new Student("Mayank Sharma", 70));
        students.add(new Student("Kushan Jigyasu", 88));
        students.add(new Student("Ansh Panwar", 60));
        students.add(new Student("Kamal Sharma", 80));

        List<String> filteredStudentNames = students.stream()
            .filter(student -> student.getMarks() > 75)
            .sorted(Comparator.comparingDouble(Student::getMarks).reversed())
            .map(Student::getName)
            .collect(Collectors.toList());

        System.out.println("Students scoring above 75% sorted by marks:");
        for (String name : filteredStudentNames) {
            System.out.println(name);
        }
    }
}
```

5. **Output**:



```
● PS D:\PBLJ>  & 'C:\Program Files\Java\jdk-17\bin\j
  orkspaceStorage\9821a2369c944fde6ad1ceda6f40c905\r
  Students scoring above 75% sorted by marks:
  Kushan Jigyasu
  Vivek Garg
  Kamal Sharma
○ PS D:\PBLJ>
```

6. **Learning Outcomes:**

- Understand the concept of lambda expressions and functional programming in Java.
- Learn how to use Java Streams for filtering, sorting, and mapping operations.
- Gain experience in working with ArrayList and Java Collections to store and process data dynamically.
- Develop proficiency in using .filter(), .sorted(), .map(), and .collect() methods in stream operations.
- Enhance skills in working with comparator functions to sort data efficiently.
- Improve understanding of real-world data processing using functional programming techniques.

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. **Objective:** Demonstrate the **use of Java Streams** to process a large dataset of products by grouping them by category, finding the most expensive product in each category, and calculating the average price of all products.

3. Algorithm:

Step 1: Initialize the Program

1. Start the program.
2. Import ArrayList, List, Map, Collectors and Comparator classes.
3. Define the product class with attributes.
    - Name
    - Category
    - Price
4. Implement a constructor to initialize these attributes.
5. Override the toString() method for formatted output.

Step 2: Create a List of Products

1. Define the main() method in the hardProductProcessor class.
2. Create a List<Product> using ArrayList<>.
3. Add multiple Product objects with sample data.

Step 3: Apply Stream Operations

1. Group products by category using .**collect(Collectors.groupingBy(Product::getCategory)).**
2. Find the most expensive product in each category using **.collect(Collectors.toMap())** combined with **.max(Comparator.comparing(Product::getPrice)).**
3. Calculate the average price of all products using **.mapToDouble(Product::getPrice).average().**

Step 4: Display the Results

1. Iterate through the grouped products and print the category-wise list.
2. Display the most expensive product in each category.
3. Print the average price of all products.

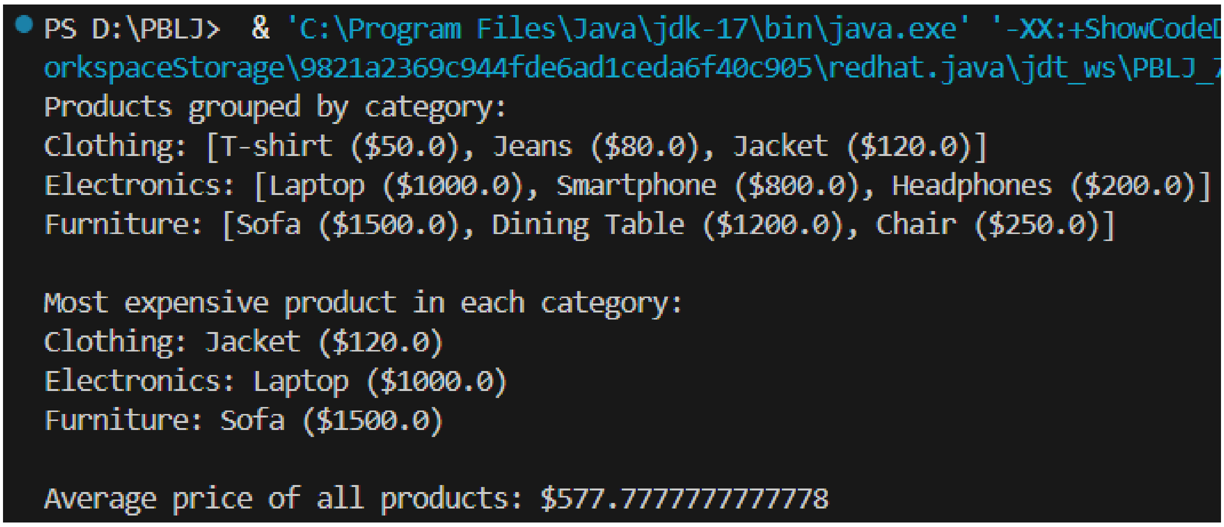
Step 5: Execute and Terminate the Program

1. Run the program.
2. Observe the grouped products, most expensive items and the average price.
3. End the execution.

4. **Code:**

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

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

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

    public String getName() {
        return name;
    }

    public String getCategory() {
        return category;
    }

    public double getPrice() {
        return price;
    }

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

public class hardProductProcessor {
    public static void main(String[] args) {
        List<Product> products = new ArrayList<>();
        products.add(new Product("Laptop", "Electronics", 1000));
        products.add(new Product("Smartphone", "Electronics", 800));
        products.add(new Product("Headphones", "Electronics", 200));
        products.add(new Product("Sofa", "Furniture", 1500));
        products.add(new Product("Dining Table", "Furniture", 1200));
        products.add(new Product("Chair", "Furniture", 250));
        products.add(new Product("T-shirt", "Clothing", 50));
        products.add(new Product("Jeans", "Clothing", 80));
        products.add(new Product("Jacket", "Clothing", 120));

        Map<String, List<Product>> productsByCategory = products.stream()
            .collect(Collectors.groupingBy(Product::getCategory));

        Map<String, Product> mostExpensiveByCategory = products.stream()
            .collect(Collectors.toMap(Product::getCategory,p -> p,(p1, p2) ->
p1.getPrice() > p2.getPrice() ? p1 : p2 ));

        double averagePrice = products.stream().mapToDouble(Product::getPrice)
            .average().orElse(0.0);

        System.out.println("Products grouped by category:");
        productsByCategory.forEach((category, productList) ->
            System.out.println(category + ": " + productList));

        System.out.println("\nMost expensive product in each category:");
        mostExpensiveByCategory.forEach((category, product) ->
            System.out.println(category + ": " + product));

        System.out.println("\nAverage price of all products: $" + averagePrice);
    }
}
```

5. **Output**:



```
● PS D:\PBLJ>  & 'C:\Program Files\Java\jdk-17\bin\java.exe' '-XX:+ShowCodeD
  orkspaceStorage\9821a2369c944fde6ad1ceda6f40c905\redhat.java\jdt_ws\PBLJ_7
  Products grouped by category:
  Clothing: [T-shirt ($50.0), Jeans ($80.0), Jacket ($120.0)]
  Electronics: [Laptop ($1000.0), Smartphone ($800.0), Headphones ($200.0)]
  Furniture: [Sofa ($1500.0), Dining Table ($1200.0), Chair ($250.0)]

  Most expensive product in each category:
  Clothing: Jacket ($120.0)
  Electronics: Laptop ($1000.0)
  Furniture: Sofa ($1500.0)

  Average price of all products: $577.7777777777778
```

6. Learning Outcomes:

- Understand the use of **Java Streams** for data processing in large datasets.
- Learn how to group elements in a collection using **Collectors.groupingBy().**
- Gain experience in finding the maximum value in a group using **Comparator.comparing().**
- Develop proficiency in working with Map and **Collectors.toMap()** for efficient data organization.
- Enhance skills in calculating statistical values like average price using **.mapToDouble().average().**
- Improve problem-solving ability in handling and manipulating collections dynamically.