Experiment -6

StudentName:Dilbag **UID:22BCS16350**
Branch: BE-CSE **Section/Group:IOT_642-B**
Semester:6th **Date of Performance:03/03/2025**
Subject Name: Project Based Learning in Java with Lab **Subject Code: 22CSH-359**

6.1.1Aim: Write a program to sort a list of Employee objects (name, age, salary) using lambda expressions..

6.1.2Objective: To implement a Java program that efficiently sorts a list of Employee objects based on name, age, and salary using lambda expressions and stream operations, demonstrating modern Java features for concise and efficient data processing.

6.1.3Code:

```
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 void display() {
        System.out.println(name + " - Age: " + age + ", Salary: " + salary);
    }
}

public class EmployeeSortLambda {
    public static void main(String[] args) {
        List<Employee> employees = new ArrayList<>();
        employees.add(new Employee("Alice", 30, 50000));
        employees.add(new Employee("Bob", 25, 60000));
```

```
    employees.add(new Employee("Charlie", 35, 55000));
    employees.add(new Employee("Alex", 28, 45000));
    employees.add(new Employee("Alex", 32, 47000));
    employees.add(new Employee("Alex", 25, 46000));
    employees.add(new Employee("David", 29, 50000));
    employees.add(new Employee("Eve", 31, 50000));
    employees.add(new Employee("Frank", 27, 50000));

    List<Employee> sortedByName = employees.stream()
        .sorted(Comparator.comparing(e -> e.name))
        .collect(Collectors.toList());

    System.out.println("Sorted by Name:");
    sortedByName.forEach(Employee::display);

    List<Employee> sortedByAge = employees.stream()
        .sorted(Comparator.comparingInt(e -> e.age))
        .collect(Collectors.toList());

    System.out.println("\nSorted by Age:");
    sortedByAge.forEach(Employee::display);
    List<Employee> sortedBySalary = employees.stream()
        .sorted((e1, e2) -> Double.compare(e2.salary, e1.salary))
        .collect(Collectors.toList());

    System.out.println("\nSorted by Salary (Descending):");
    sortedBySalary.forEach(Employee::display);
    List<Employee> sortedByNameThenAge = employees.stream()
        .sorted(Comparator.comparing((Employee e) -> e.name)
            .thenComparingInt(e -> e.age))
        .collect(Collectors.toList());

    System.out.println("\nSorted by Name, then Age:");
    sortedByNameThenAge.forEach(Employee::display);
    List<Employee> sortedBySalaryThenName = employees.stream()
        .sorted(Comparator.comparingDouble((Employee e) -> e.salary)
            .thenComparing(e -> e.name))
        .collect(Collectors.toList());

    System.out.println("\nSorted by Salary, then Name:");
    sortedBySalaryThenName.forEach(Employee::display);
}
```

6.1.4 Output:

```
Sorted by Name:
Alex - Age: 28, Salary: 45000.0
Alex - Age: 32, Salary: 47000.0
Alex - Age: 25, Salary: 46000.0
Alice - Age: 30, Salary: 50000.0
Bob - Age: 25, Salary: 60000.0
Charlie - Age: 35, Salary: 55000.0
David - Age: 29, Salary: 50000.0
Eve - Age: 31, Salary: 50000.0
Frank - Age: 27, Salary: 50000.0

Sorted by Age:
Bob - Age: 25, Salary: 60000.0
Alex - Age: 25, Salary: 46000.0
Frank - Age: 27, Salary: 50000.0
Alex - Age: 28, Salary: 45000.0
David - Age: 29, Salary: 50000.0
Alice - Age: 30, Salary: 50000.0
Eve - Age: 31, Salary: 50000.0
Alex - Age: 32, Salary: 47000.0
Charlie - Age: 35, Salary: 55000.0
```

```
Sorted by Salary (Descending):
Bob - Age: 25, Salary: 60000.0
Charlie - Age: 35, Salary: 55000.0
Alice - Age: 30, Salary: 50000.0
David - Age: 29, Salary: 50000.0
Eve - Age: 31, Salary: 50000.0
Frank - Age: 27, Salary: 50000.0
Alex - Age: 32, Salary: 47000.0
Alex - Age: 25, Salary: 46000.0
Alex - Age: 28, Salary: 45000.0

Sorted by Name, then Age:
Alex - Age: 25, Salary: 46000.0
Alex - Age: 28, Salary: 45000.0
Alex - Age: 32, Salary: 47000.0
Alice - Age: 30, Salary: 50000.0
Bob - Age: 25, Salary: 60000.0
Charlie - Age: 35, Salary: 55000.0
David - Age: 29, Salary: 50000.0
Eve - Age: 31, Salary: 50000.0
Frank - Age: 27, Salary: 50000.0
```

```
Sorted by Salary, then Name:
Alex - Age: 28, Salary: 45000.0
Alex - Age: 25, Salary: 46000.0
Alex - Age: 32, Salary: 47000.0
Alice - Age: 30, Salary: 50000.0
David - Age: 29, Salary: 50000.0
Eve - Age: 31, Salary: 50000.0
Frank - Age: 27, Salary: 50000.0
Charlie - Age: 35, Salary: 55000.0
Bob - Age: 25, Salary: 60000.0
```

6.2.1Aim: Create a program to use lambda expressions and stream operations to filter students scoring above 75%, sort them by marks, and display their names

6.2.2Objective: To implement a Java program that filters students scoring above 75%, sorts them in descending order using lambda expressions and Stream API, and efficiently displays the results.

6.2.3 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 void display() {
        System.out.println(name + " - Marks: " + marks);
    }
}

public class StudentFilterSort {
    public static void main(String[] args) {
        List<Student> students = Arrays.asList(
            new Student("Alice", 80),
            new Student("Bob", 72),
            new Student("Charlie", 90),
            new Student("David", 65),
            new Student("Eve", 85),
            new Student("Frank", 65)
        );

        List<Student> filteredSortedStudents = students.stream()
                .filter(s -> s.marks > 75)
                .sorted(Comparator.comparingDouble((Student s) -> s.marks).reversed()
                        .thenComparing(s -> s.name))
```

```
                .collect(Collectors.toList());

        System.out.println("Students who scored above 75% (Sorted by Marks):");
        if (filteredSortedStudents.isEmpty()) {
            System.out.println("No students scored above 75%");
        } else {
            filteredSortedStudents.forEach(Student::display);
        }

        runTestCases();
    }

    public static void runTestCases() {
        System.out.println("\n===== Running Test Cases =====");
        System.out.println("\nTest Case 1: Normal Case");
        testFilterSort(Arrays.asList(
                new Student("Alice", 80),
                new Student("Bob", 72),
                new Student("Charlie", 90),
                new Student("David", 65),
                new Student("Eve", 85)
        ));

        System.out.println("\nTest Case 2: All Below 75%");
        testFilterSort(Arrays.asList(
                new Student("Bob", 70),
                new Student("David", 60),
                new Student("Frank", 65)
        ));
        System.out.println("\nTest Case 3: Same Marks");
        testFilterSort(Arrays.asList(
                new Student("Alice", 80),
                new Student("Bob", 80),
                new Student("Charlie", 85)
        ));

        System.out.println("\nTest Case 4: Single Student Above 75%");
        testFilterSort(Arrays.asList(
                new Student("Alice", 60),
                new Student("Bob", 50),
                new Student("Charlie", 90)
        ));
```

```
    }

    public static void testFilterSort(List<Student> students) {
        List<Student> result = students.stream()
                .filter(s -> s.marks > 75)
                .sorted(Comparator.comparingDouble((Student s) -> s.marks).reversed()
                        .thenComparing(s -> s.name))
                .collect(Collectors.toList());

        if (result.isEmpty()) {
            System.out.println("No students scored above 75%");
        } else {
            result.forEach(Student::display);
        }
    }
}
```

6.2.4Output:

```
Students who scored above 75% (Sorted by Marks):
Charlie - Marks: 90.0
Eve - Marks: 85.0
Alice - Marks: 80.0

===== Running Test Cases =====

Test Case 1: Normal Case
Charlie - Marks: 90.0
Eve - Marks: 85.0
Alice - Marks: 80.0

Test Case 2: All Below 75%
No students scored above 75%

Test Case 3: Same Marks
Charlie - Marks: 85.0
Alice - Marks: 80.0
Bob - Marks: 80.0

Test Case 4: Single Student Above 75%
Charlie - Marks: 90.0
```

6.3.1Aim: 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.

6.3.2Objective: The objective of this Java program is to process a large product dataset using the Streams API by grouping products by category, finding the most expensive product in each category, and calculating the average price efficiently.

6.3.3Code:

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

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 void display() {
        System.out.println(name + " (" + category + ") - Price: $" + price);
    }
}

public class ProductProcessor {
    public static void main(String[] args) {
        List<Product> products = Arrays.asList(
            new Product("Laptop", "Electronics", 1200),
            new Product("Phone", "Electronics", 800),
            new Product("TV", "Electronics", 1500),
            new Product("T-Shirt", "Clothing", 40),
            new Product("Jeans", "Clothing", 60),
            new Product("Sneakers", "Footwear", 120),
            new Product("Boots", "Footwear", 120)
        );

        processProducts(products);
```

```
        runTestCases();
    }

    public static void processProducts(List<Product> products) {

        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()
                .collect(Collectors.averagingDouble(p -> p.price));


        System.out.println("=== Grouped Products by Category ===");
        groupedByCategory.forEach((category, productList) -> {
            System.out.println(category + ": " + productList.stream()
                    .map(p -> p.name)
                    .collect(Collectors.joining(", ")));
        });

        System.out.println("\n=== Most Expensive Product in Each Category ===");
        mostExpensiveByCategory.forEach((category, product) ->
            System.out.println(category + ": " + (product.isPresent() ? product.get().name + " - $" +
product.get().price : "No products")));

        System.out.println("\n=== Average Price of All Products ===");
        System.out.printf("Average Price: $%.2f%n", averagePrice);
    }

    public static void runTestCases() {
        System.out.println("\n===== Running Test Cases =====");

        // Test Case 1: Normal Case
        System.out.println("\nTest Case 1: Normal Case");
        processProducts(Arrays.asList(
                new Product("Laptop", "Electronics", 1200),
                new Product("Phone", "Electronics", 800),
                new Product("TV", "Electronics", 1500),
```

```
            new Product("T-Shirt", "Clothing", 40),
            new Product("Jeans", "Clothing", 60),
            new Product("Sneakers", "Footwear", 120),
            new Product("Boots", "Footwear", 120)
        ));

        // Test Case 2: Single Category Only
        System.out.println("\nTest Case 2: Single Category Only");
        processProducts(Arrays.asList(
            new Product("Laptop", "Electronics", 1200),
            new Product("Phone", "Electronics", 800),
            new Product("TV", "Electronics", 1500)
        ));

        // Test Case 3: Same Price in a Category
        System.out.println("\nTest Case 3: Same Price in a Category");
        processProducts(Arrays.asList(
            new Product("Sneakers", "Footwear", 120),
            new Product("Boots", "Footwear", 120)
        ));

        // Test Case 4: Only One Product
        System.out.println("\nTest Case 4: Only One Product");
        processProducts(Arrays.asList(
            new Product("Laptop", "Electronics", 1200)
        ));

        // Test Case 5: Empty List
        System.out.println("\nTest Case 5: Empty List");
        processProducts(Collections.emptyList());
    }
}
```

6.3.4Output:

```
=== Grouped Products by Category ===
Clothing: T-Shirt, Jeans
Footwear: Sneakers, Boots
Electronics: Laptop, Phone, TV

=== Most Expensive Product in Each Category ===
Clothing: Jeans - $60.0
Footwear: Sneakers - $120.0
Electronics: TV - $1500.0

=== Average Price of All Products ===
Average Price: $548.57
```

```
===== Running Test Cases =====

Test Case 1: Normal Case
=== Grouped Products by Category ===
Clothing: T-Shirt, Jeans
Footwear: Sneakers, Boots
Electronics: Laptop, Phone, TV

=== Most Expensive Product in Each Category ===
Clothing: Jeans - $60.0
Footwear: Sneakers - $120.0
Electronics: TV - $1500.0

=== Average Price of All Products ===
Average Price: $548.57

Test Case 2: Single Category Only
=== Grouped Products by Category ===
Electronics: Laptop, Phone, TV

=== Most Expensive Product in Each Category ===
Electronics: TV - $1500.0

=== Average Price of All Products ===
Average Price: $1166.67
```

```
Test Case 3: Same Price in a Category
=== Grouped Products by Category ===
Footwear: Sneakers, Boots

=== Most Expensive Product in Each Category ===
Footwear: Sneakers - $120.0

=== Average Price of All Products ===
Average Price: $120.00

Test Case 4: Only One Product
=== Grouped Products by Category ===
Electronics: Laptop

=== Most Expensive Product in Each Category ===
Electronics: Laptop - $1200.0

=== Average Price of All Products ===
Average Price: $1200.00

Test Case 5: Empty List
=== Grouped Products by Category ===

=== Most Expensive Product in Each Category ===

=== Average Price of All Products ===
Average Price: $0.00
```

Learning Outcomes:

1. Lambda Expressions & Functional Programming – Utilize concise and readable lambda expressions for sorting, filtering, and processing data.
2. Streams API for Efficient Data Handling – Learn to filter, sort, group, and aggregate data using Streams and Collectors.
3. Sorting& Filtering with Streams – Implement sorting (ascending/descending) and filtering conditions dynamically.
4. Grouping& Aggregation – Use Collectors.groupingBy(), Collectors.maxBy(), and Collectors.averagingDouble() for data analysis.
5. Handling Edge Cases – Manage empty lists, duplicate values, and single-element scenarios gracefully.