Experiment 6.1

Student Name: SAHIL **UID: 22BCS14873**
Branch: CSE **Section: 22BCS_IOT-642-A**
Semester: 6th **DOP: 04/03/25**
Subject: PBLJ **Subject Code:22CSH-359**

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

Objective: The goal of this Java program is to demonstrate sorting a list of Employee objects using lambda expressions and the Comparator interface.

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 EmployeeSorter {
    public static void main(String[] args) {
        List<Employee> employees = new ArrayList<>();
        employees.add(new Employee("SAHIL", 21, 70000));
        employees.add(new Employee("SAMAY", 30, 60000));
        employees.add(new Employee("ROHIT", 25, 90000));


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


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

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

Output:

```
Sorted by name:
Employee{name='ROHIT', age=25, salary=90000.0}
Employee{name='SAHIL', age=21, salary=70000.0}
Employee{name='SAMAY', age=30, salary=60000.0}

Sorted by age:
Employee{name='SAHIL', age=21, salary=70000.0}
Employee{name='ROHIT', age=25, salary=90000.0}
Employee{name='SAMAY', age=30, salary=60000.0}

Sorted by salary:
Employee{name='SAMAY', age=30, salary=60000.0}
Employee{name='SAHIL', age=21, salary=70000.0}
Employee{name='ROHIT', age=25, salary=90000.0}


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

Learning Outcomes:

- Understand how to define and use lambda expressions in Java.
- Learn how to sort collections using Comparator and lambda expressions.
- Gain experience in working with lists and custom objects in Java.
- Develop skills in utilizing Java Streams and functional programming techniques.

Experiment 6.2

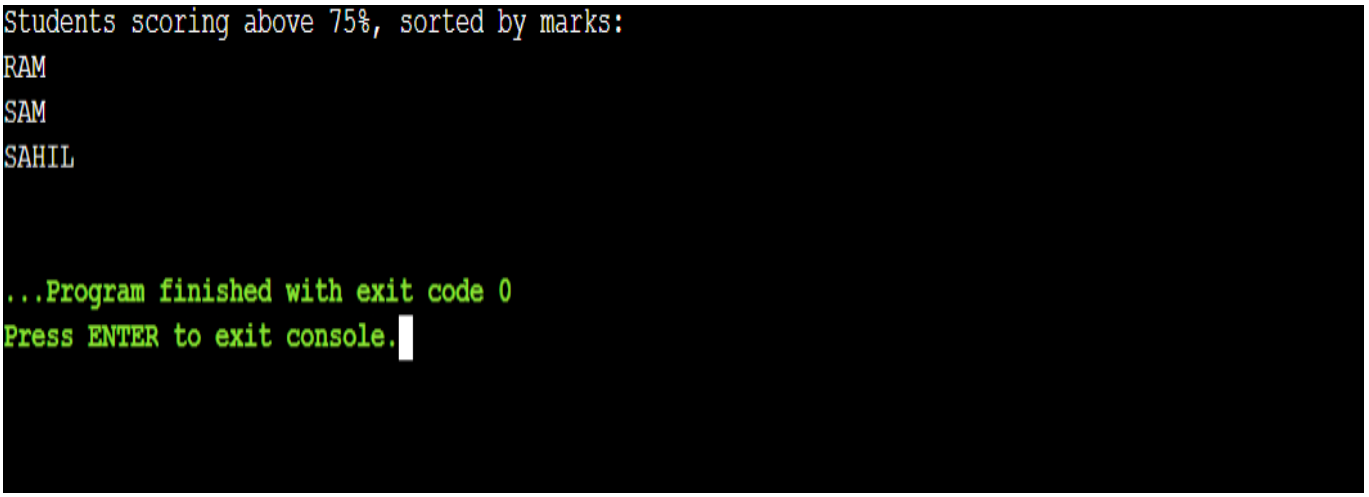
Aim: Create a program to use lambda expressions and stream operations to filter students scoring above 75%, sort them by marks, and display their names.

Objective: The objective is to demonstrate the use of lambda expressions and stream operations in Java.

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 getName() {
        return name;
    }

    public double getMarks() {
        return marks;
    }
}

public class StudentFilterSort {
    public static void main(String[] args) {
        List<Student> students = Arrays.asList(
            new Student("SAHIL", 80),
            new Student("ROHAN", 70),
            new Student("RAM", 90),
            new Student("ROHIT", 60),
            new Student("SAM", 85)
        );

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

        System.out.println("Students scoring above 75%, sorted by marks:");
        topStudents.forEach(System.out::println);
    }
}
```

Output:



Learning Outcomes:

- Understand the use of Java Streams for data processing.
- Learn how to filter, sort, and collect data using lambda expressions.
- Gain experience with functional programming concepts in Java.
- Develop skills in working with lists and custom objects efficiently.

Experiment 6.3

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.

Objective: The objective of this Java application is to demonstrate the use of Java Streams for processing large datasets efficiently.

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 String getCategory() {
        return category;
    }

    public double getPrice() {
        return price;
    }

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

public class ProductProcessor {
    public static void main(String[] args) {
        List<Product> products = Arrays.asList(
            new Product("Laptop", "Electronics", 1200.00),
            new Product("Phone", "Electronics", 800.00),
            new Product("TV", "Electronics", 1500.00),
            new Product("Sofa", "Furniture", 700.00),
            new Product("Table", "Furniture", 300.00),
            new Product("Chair", "Furniture", 150.00)
        );

        Map<String, List<Product>> productsByCategory = products.stream()
            .collect(Collectors.groupingBy(Product::getCategory));
        System.out.println("Products grouped by category:");
        productsByCategory.forEach((category, productList) ->
```

```
            System.out.println(category + ": " + productList));

        Map<String, Optional<Product>> mostExpensiveByCategory = products.stream()
            .collect(Collectors.groupingBy(
                Product::getCategory,
                Collectors.maxBy(Comparator.comparingDouble(Product::getPrice))
            ));
        System.out.println("\nMost expensive product in each category:");
        mostExpensiveByCategory.forEach((category, product) ->
            System.out.println(category + ": " + product.orElse(null)));

        double averagePrice = products.stream()
            .mapToDouble(Product::getPrice)
            .average()
            .orElse(0.0);
        System.out.println("\nAverage price of all products: $" + averagePrice);
    }
}
```

Output:

```
Products grouped by category:
Electronics: [Laptop ($1200.0), Phone ($800.0), TV ($1500.0)]
Furniture: [Sofa ($700.0), Table ($300.0), Chair ($150.0)]

Most expensive product in each category:
Electronics: TV ($1500.0)
Furniture: Sofa ($700.0)

Average price of all products: $775.0


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

Learning Outcomes:

- Understand how to use Java Streams for large-scale data processing.
- Learn how to group data using Collectors.
- Implement aggregation functions such as max and average on grouped data.
- Gain experience in functional programming techniques in Java.