



Experiment 6.1

Student Name: Satyam Kr. Thakur

UID: 22BCS14812

Branch: CSE

Section/Group: 22BCS_IOT-640/A

Semester: 6th

Date of Performance: 03/03/25

Subject Name: PBLJ

Subject Code: 22CSH-359

1. **Aim:** Write a program to sort a list of Employee objects (name, age, salary) using lambda expressions.
2. **Objective :** The objective of this program is to demonstrate how to sort a list of Employee objects based on different attributes (name, age, and salary) using lambda expressions in Java. This will help in understanding how to use Java's Comparator with lambda functions to simplify sorting operations and enhance code readability.

3. Implementation/Code:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

class Employee {
    private String name;
    private int age;
    private double salary;

    public Employee(String name, int age, double salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

```
    public double getSalary() {
        return salary;
    }

    @Override
    public String toString() {
        return "Employee{" +
            "name=\"" + name + "\"" +
            ", age=" + age +
            ", salary=" + salary +
            '}';
    }
}

public class EmployeeSortingExample {
    public static void main(String[] args) {
        List<Employee> employees = new ArrayList<>();
        employees.add(new Employee("John Doe", 30, 50000));
        employees.add(new Employee("Jane Smith", 25, 60000));
        employees.add(new Employee("Michael Johnson", 35, 45000));

        System.out.println("Employees before sorting:");
        employees.forEach(System.out::println);

        // Sort employees by name (ascending)
        Collections.sort(employees, Comparator.comparing(Employee::getName));

        System.out.println("\nEmployees after sorting by name (ascending):");
        employees.forEach(System.out::println);

        // Sort employees by age (descending)
        Collections.sort(employees,
            Comparator.comparingInt(Employee::getAge).reversed());

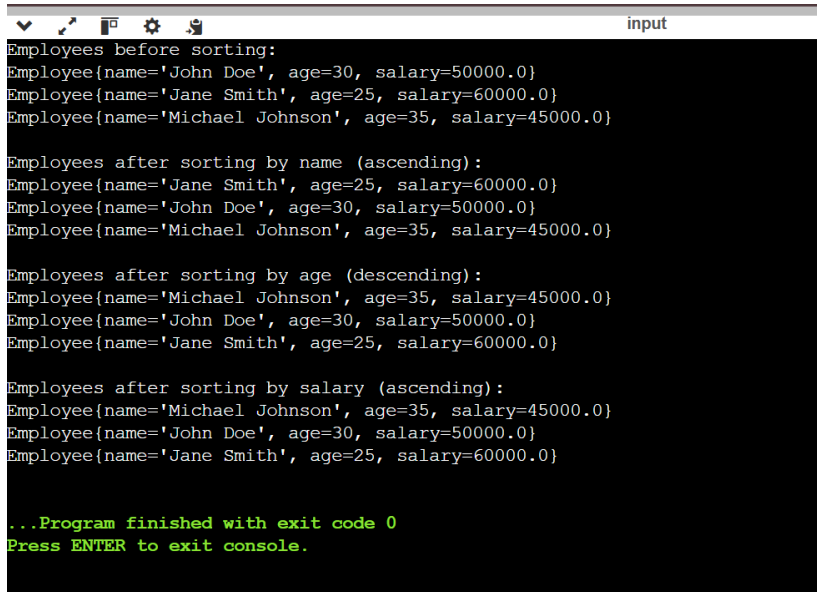
        System.out.println("\nEmployees after sorting by age (descending):");
        employees.forEach(System.out::println);

        // Sort employees by salary (ascending)
        Collections.sort(employees, Comparator.comparingDouble(Employee::getSalary));

        System.out.println("\nEmployees after sorting by salary (ascending):");
```

```
        employees.forEach(System.out::println);  
    }  
}
```

4. Output:



```
Employees before sorting:  
Employee{name='John Doe', age=30, salary=50000.0}  
Employee{name='Jane Smith', age=25, salary=60000.0}  
Employee{name='Michael Johnson', age=35, salary=45000.0}  
  
Employees after sorting by name (ascending):  
Employee{name='Jane Smith', age=25, salary=60000.0}  
Employee{name='John Doe', age=30, salary=50000.0}  
Employee{name='Michael Johnson', age=35, salary=45000.0}  
  
Employees after sorting by age (descending):  
Employee{name='Michael Johnson', age=35, salary=45000.0}  
Employee{name='John Doe', age=30, salary=50000.0}  
Employee{name='Jane Smith', age=25, salary=60000.0}  
  
Employees after sorting by salary (ascending):  
Employee{name='Michael Johnson', age=35, salary=45000.0}  
Employee{name='John Doe', age=30, salary=50000.0}  
Employee{name='Jane Smith', age=25, salary=60000.0}  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

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 :

The objective of this program is to demonstrate the use of lambda expressions and stream operations in Java to efficiently filter, sort, and display student data. Specifically, we will:

1. Filter students who have scored above 75%.
2. Sort the filtered students by their marks in descending order.
3. Display the names of the top-performing students.

This approach leverages functional programming features in Java (Lambda Expressions & Streams) to write clean and efficient code.

3. Implementation /Code :

```
import java.util.ArrayList;
import java.util.List;
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;
    }

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

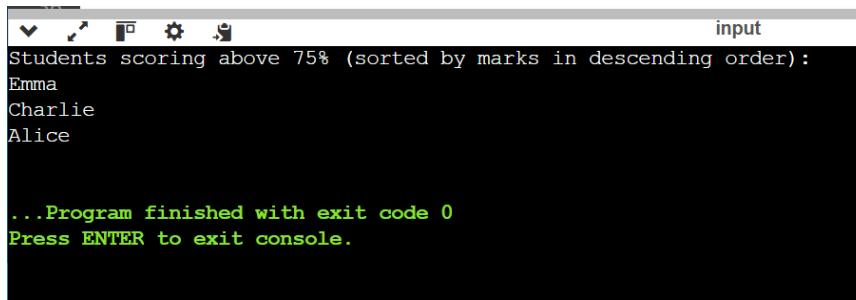
public class StudentFiltering {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student("Alice", 78.5));
        students.add(new Student("Bob", 67.0));
        students.add(new Student("Charlie", 85.2));
        students.add(new Student("David", 72.8));
        students.add(new Student("Emma", 90.3));

        System.out.println("Students scoring above 75% (sorted by marks in
        descending order):");

        // Using stream operations to filter, sort, and display students
```

```
        students.stream()
            .filter(s -> s.getMarks() > 75) // Filtering students with marks > 75
            .sorted((s1, s2) -> Double.compare(s2.getMarks(), s1.getMarks())) //
Sorting in descending order
            .map(Student::getName) // Extracting only names
            .forEach(System.out::println); // Displaying names
    }
}
```

4. Output :



```
input
Students scoring above 75% (sorted by marks in descending order):
Emma
Charlie
Alice

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

Experiment 6.3

1. **Aim :** Write a Java program to process a large dataset of products using streams.
2. **Objective :** Perform operations such as grouping products by category, finding the most expensive product in each category, and calculating the average price of all products.

3. Implementation /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 ProductProcessing {
    public static void main(String[] args) {
        List<Product> products = Arrays.asList(
            new Product("Laptop", "Electronics", 1200.00),
            new Product("Smartphone", "Electronics", 800.00),
            new Product("TV", "Electronics", 1500.00),
            new Product("Sofa", "Furniture", 700.00),
            new Product("Dining Table", "Furniture", 1200.00),
            new Product("Bed", "Furniture", 2000.00),
            new Product("Jeans", "Clothing", 50.00),
            new Product("T-Shirt", "Clothing", 20.00),
            new Product("Jacket", "Clothing", 100.00)
        );

        // 1. Group products by category
        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);
    });

    // 2. Find the most expensive product in each category
    Map<String, Optional<Product>> mostExpensiveProductByCategory =
products.stream()
        .collect(Collectors.groupingBy(
            Product::getCategory,

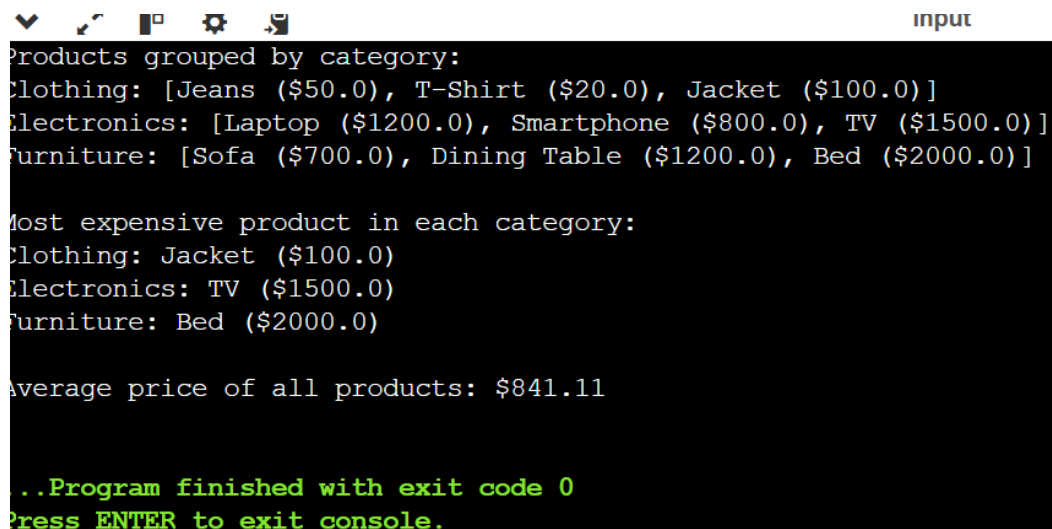
Collectors.maxBy(Comparator.comparingDouble(Product::getPrice))
        ));

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

    // 3. Calculate the average price of all products
    double averagePrice = products.stream()
        .collect(Collectors.averagingDouble(Product::getPrice));

    System.out.println("\nAverage price of all products: $" +
String.format("%.2f", averagePrice));
    }
}
```

4. Output :



```
input
Products grouped by category:
Clothing: [Jeans ($50.0), T-Shirt ($20.0), Jacket ($100.0)]
Electronics: [Laptop ($1200.0), Smartphone ($800.0), TV ($1500.0)]
Furniture: [Sofa ($700.0), Dining Table ($1200.0), Bed ($2000.0)]

Most expensive product in each category:
Clothing: Jacket ($100.0)
Electronics: TV ($1500.0)
Furniture: Bed ($2000.0)

Average price of all products: $841.11

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



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

5. Learning Outcomes :

- Learn about Serialization and Deserialization.
- Understanding File Handling in Java.
- Exceptional Handling in Java.
- Enhancing Problem-Solving Skills.



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.