



DEPARTMENT OF

COMPUTER SCIENCE & ENGINEERING

Experiment 6.1

Student Name: Manikanteswara Reddy

UID:22BCS14944

Branch: CSE

Section:643/A

Semester: 6th

DOP:03/03/25

Subject: PBLJ

Subject Code:22CSH-359

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

Objective: The aim of this program is to demonstrate the use of lambda expressions and Java Stream API to efficiently sort a list of Employee objects based on different attributes — name, age, and salary. This approach simplifies sorting operations with concise and readable code, making it ideal for handling large datasets. Let me know if you'd like me to add more explanations or modify the program!.

Algorithm:

1. Define the Employee Class:

- Create fields: name, age, and salary.
- Add a constructor to initialize these fields.
- Override the toString() method for readable output.

2. Create a List of Employee Objects:

- Use Arrays.asList() to quickly create sample data.

3. Sort by Name:

- Use stream() on the list.
- Apply sorted() with Comparator.comparing() on the name field.
- Use forEach() with System.out::println to display results.

4. Sort by Age:

- Repeat the sorting process, using Comparator.comparingInt() on the age field.

5. Sort by Salary:

- Again, use stream(), sorted(), and Comparator.comparingDouble() on the salary field.

6. Display Sorted Results:

- Print the sorted list for each criterion: name, age, and salary

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

public class EmployeeSorting {
    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(
            new Employee("Alice", 30, 50000),
            new Employee("Bob", 25, 60000),
            new Employee("Charlie", 35, 55000)
        );

        System.out.println("Sorting by name:");
        employees.stream()
            .sorted(Comparator.comparing(e -> e.name))
            .forEach(System.out::println);

        System.out.println("\nSorting by age:");
        employees.stream()
            .sorted(Comparator.comparingInt(e -> e.age))
            .forEach(System.out::println);

        System.out.println("\nSorting by salary:");
        employees.stream()
            .sorted(Comparator.comparingDouble(e -> e.salary))
            .forEach(System.out::println);
    }
}
```

Output:

```
Sorting by name:
Employee{name='Alice', age=30, salary=50000.00}
Employee{name='Bob', age=25, salary=60000.00}
Employee{name='Charlie', age=35, salary=55000.00}

Sorting by age:
Employee{name='Bob', age=25, salary=60000.00}
Employee{name='Alice', age=30, salary=50000.00}
Employee{name='Charlie', age=35, salary=55000.00}

Sorting by salary:
Employee{name='Alice', age=30, salary=50000.00}
Employee{name='Charlie', age=35, salary=55000.00}
Employee{name='Bob', age=25, salary=60000.00}

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

Learning Outcomes:

- **Understanding Lambda Expressions:** Learn how to write concise and readable code using Java's lambda expressions for functional-style programming.
- **Implementing Sorting:** Learn how to sort objects based on different attributes like name, age, and salary using Comparator and lambda functions.
- **Enhancing Code Efficiency:** Develop skills to write less boilerplate code with more efficiency and clarity.
- **Handling Large Datasets:** Learn techniques to process and manage large datasets with optimal performance using streams..

Experiment 6.2

1.Aim: Develop Java programs using lambda expressions and stream operations for sorting, filtering, and processing large datasets efficiently. 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:

- a. Use Java lambda expressions and Stream API to efficiently process a list of student objects.
- b. Filter students who scored above 75%.
- c. Sort the filtered students by their marks in ascending order.
- d. Display only the names of the sorted students.

3.Algorithm:

1. Define the Student Class:

- Fields: name and marks.
- Constructor to initialize these fields.
- Override toString() for readable output (optional).

2. Create a List of Students:

- Use Arrays.asList() to populate sample student data.

3. Filter Students:

- Use stream() on the student list.
- Apply filter() to include only students with marks above 75%.

4. Sort Students by Marks:

- Use sorted() with Comparator.comparingDouble() on the marks field.

5. Extract and Display Names:

- Use map() to extract only the name field.
- Use forEach() with System.out::println to display the names.

4. Implementation Code:

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

class Student {
    String name;
    double marks;

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

    @Override
    public String toString() {
        return
String.format("Student {name='%s',
marks=%.2f}", name, marks);
    }
}

public class StudentFilterSort {
    public static void main(String[]
args) {
        List<Student> students =
Arrays.asList(
            new Student("John", 72.5),
            new Student("Alice", 85.0),
            new Student("Bob", 78.5),
            new Student("Daisy", 91.0),
            new Student("Eve", 67.0)
        );

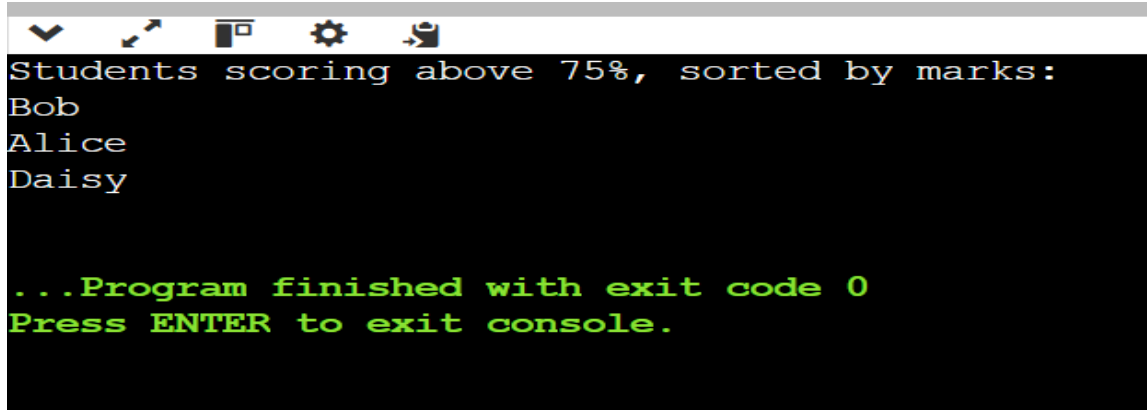
        System.out.println("Students
scoring above 75%, sorted by
marks:");
        students.stream()
            .filter(s -> s.marks > 75)

            .sorted(Comparator.comparingDouble(
s -> s.marks))
    }
```

```
.map(s -> s.name)
```

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

5.Output



```
Students scoring above 75%, sorted by marks:  
Bob  
Alice  
Daisy  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

6.Learning Outcomes:

- Mastering Lambda Expressions:** Understand how to write clean and efficient code using Java's lambda expressions.
- Using the Stream API:** Learn how to process collections efficiently with stream operations.
- Filtering Data:** Gain the ability to filter objects based on specific conditions (like marks > 75%).
- Sorting Collections:** Practice sorting objects using comparators and stream operations.

Experiment 6.3

1. **Aim:** Develop Java programs using lambda expressions and stream operations for sorting, filtering, and processing large datasets efficiently. 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:**
 - a. Use Java lambda expressions and the Stream API to efficiently process a large dataset of products.
 - b. Group products by their category.
 - c. Find the most expensive product in each category.
 - d. Calculate the average price of all products.
3. **Algorithm:**
 1. **Define the Product Class:**
 - Fields: name, category, price.
 - Constructor to initialize the fields.
 - Override toString() for readable output (optional).
 2. **Create a List of Products:**
 - Use Arrays.asList() to add sample product data.
 3. **Group Products by Category:**
 - Use stream() and collect(Collectors.groupingBy()) to group products by their category.
 4. **Find the Most Expensive Product in Each Category:**
 - Use collect(Collectors.groupingBy()) with Collectors.maxBy() to get the most expensive product per category.
 5. **Calculate the Average Price of All Products:**
 - Use mapToDouble() and average() on the product list to calculate the average price.
 6. **Display Results:**
 - Print grouped products, the most expensive product in each category, and the average price.

4.Implementation Code:

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

```
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;  
    }  
}
```

```
}

@Override
public String toString() {
    return
String.format("Product{name='
%s', category='%s',
price=%.2f}", name, category,
price);
}
}

public class
ProductDataProcessing {
    public static void
main(String[] args) {
        List<Product> products =
Arrays.asList(
            new
Product("Laptop",
"Electronics", 1000.0),
            new
Product("Smartphone",
"Electronics", 800.0),
            new
Product("Headphones",
"Electronics", 150.0),
            new Product("Sofa",
"Furniture", 700.0),
            new Product("Chair",
"Furniture", 150.0),
            new Product("Table",
"Furniture", 300.0)
        );
        // Grouping products by
category

System.out.println("\nProducts
grouped by category:");
        Map<String,
List<Product>>
productsByCategory =
products.stream()
```



```
.collect(Collectors.groupingBy(  
p -> p.category));
```

```
productsByCategory.forEach((c  
ategory, productList) -> {
```

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

```
// Finding the most  
expensive product in each  
category
```

```
System.out.println("\nMost  
expensive product in each  
category:");  
products.stream()
```

```
.collect(Collectors.groupingBy(  
p -> p.category,
```

```
Collectors.maxBy(Comparator.  
comparingDouble(p -> p.price))  
))
```

```
.forEach((category,  
product) ->  
System.out.println(category + ":  
" + product.orElse(null)));
```

```
// Calculating the average  
price of all products
```

```
double averagePrice =  
products.stream()  
.mapToDouble(p ->  
p.price)  
.average()  
.orElse(0.0);
```

```
System.out.printf("\nAverage  
price of all products: %.2f\n",  
averagePrice);  
}
```

}

Output:

```
input

Products grouped by category:
Electronics: [Product{name='Laptop', category='Electronics', price=1000.00}, Product{name='Smartphone', category='Electronics', price=800.00}, Product{name='Headphones', category='Electronics', price=150.00}]
Furniture: [Product{name='Sofa', category='Furniture', price=700.00}, Product{name='Chair', category='Furniture', price=150.00}, Product{name='Table', category='Furniture', price=300.00}]

Most expensive product in each category:
Electronics: Product{name='Laptop', category='Electronics', price=1000.00}
Furniture: Product{name='Sofa', category='Furniture', price=700.00}

Average price of all products: 516.67

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

5. Learning Outcomes:

- Mastering Lambda Expressions:** Gain efficiency in writing clean and functional Java code.
- Working with the Stream API:** Learn to handle large datasets with streams for better performance and readability.
- Grouping Data:** Understand how to group objects by their attributes using `Collectors.groupingBy()`.
- Finding Maximum Values:** Learn to use `Collectors.maxBy()` to find the most expensive product in each category.