



# DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

## Experiment 6

**Student Name: Jaskaran Singh**

**UID: 22BCS15829**

**Branch: BE/CSE**

**Section/Group: IOT\_639/B**

**Semester: 6<sup>th</sup>**

**Date of Performance: 28/02/25**

**Subject Name: Project based learning in Java**

**Subject Code: 22CSH- 359**

### **Easy Level**

- 1. Aim:** Write a program to sort a list of Employee objects (name, age, salary) using lambda expressions.
- 2. Objective:** The objective of this Java program is to demonstrate how to sort a list of Employee objects based on different attributes (name, age, salary) using **lambda expressions** and the Comparator interface

### **3. Implementation/Code:**

```
import java.util.*;

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

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

```
@Override
public String toString() {
    return name + " - Age: " + age + ", Salary: " + salary;
}
}

public class EmployeeSorting {
    public static void main(String[] args)
    { List<Employee> employees = new ArrayList<>();
      employees.add(new Employee("Akshita", 20, 50000));
      employees.add(new Employee("Khushi", 22, 60000));
      employees.add(new Employee("Harshit", 23, 55000));

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

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

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

## 4. Output:

```
PS C:\Users\harsh\OneDrive\Documents\Java Sem 6> cd "c:\
ng }
Sorted by name:
Akshita - Age: 20, Salary: 50000.0
Harshit - Age: 23, Salary: 55000.0
Khushi - Age: 22, Salary: 60000.0

Sorted by age:
Akshita - Age: 20, Salary: 50000.0
Khushi - Age: 22, Salary: 60000.0
Harshit - Age: 23, Salary: 55000.0

Sorted by salary:
Akshita - Age: 20, Salary: 50000.0
Harshit - Age: 23, Salary: 55000.0
Khushi - Age: 22, Salary: 60000.0
PS C:\Users\harsh\OneDrive\Documents\Java Sem 6>
```

## 5. Learning Outcomes:

- Learnt about Comparator interface
- Efficient Data Handling
- Learn to handle user input from the command line.
- Looping and Computation.
- Understanding Java Sorting and Filteration.

## Medium Level

- 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 Java program is to demonstrate the use of **lambda expressions** and **stream operations** to efficiently process and manipulate collections.

### 3. Implementation/Code:

```
import java.util.*;
import java.util.stream.Collectors;
class Student {
    String name;
    double marks;
    Student(String name, double marks)
    { this.name = name;
      this.marks = marks;
    }
    @Override
    public String toString() {
        return name + " - Marks: " + marks;
    }
}
```

```
public class StudentFiltering {  
    public static void main(String[] args)  
    {  
        List<Student> students = new ArrayList<>();  
        students.add(new Student("Akshita", 85));  
        students.add(new Student("Harshit", 70));  
        students.add(new Student("Khushi", 90));  
        students.add(new Student("Shrey", 60));  
        students.add(new Student("Divesh", 80));  
        // Filtering students scoring above 75% and sorting by marks  
        List<Student> filteredStudents = students.stream()  
            .filter(student -> student.marks > 75)  
            .sorted(Comparator.comparingDouble(student -> student.marks))  
            .collect(Collectors.toList());  
        System.out.println("Students scoring above 75% (sorted by marks):");  
        filteredStudents.forEach(student -> System.out.println(student.name));  
    }  
}
```

#### 4. Output:

```
PS C:\Users\harsh\OneDrive\Documents\Java Sem 6> cd "c:\U  
ring }  
Students scoring above 75% (sorted by marks):  
Divesh  
Akshita  
Khushi  
PS C:\Users\harsh\OneDrive\Documents\Java Sem 6>
```



# **DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

Discover. Learn. Empower.

## **5. Learning Outcomes:**

- Understanding Java streams for Data Processing.
- Implement key-value storage.
- Add and retrieve elements dynamically without predefined limits.
- Use Scanner to take user input and process it efficiently.

## Hard Level

**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:** The objective of this Java program is to demonstrate the use of **Java Streams** to efficiently process a large dataset of products.

### 3. Implementation/Code:

```
import java.util.*;

import java.util.stream.Collectors;

class Product {

    String name;

    String category;

    double price;

    Product(String name, String category, double price)

    { this.name = name;

      this.category = category;

      this.price = price;

    }
```



# DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

@Override

```
public String toString() {
```

```
    return name + " - Category: " + category + ", Price: " + price;
```

```
}
```

```
}
```

```
public class ProductProcessing {
```

```
    public static void main(String[] args)
```

```
    { List<Product> products = Arrays.asList(
```

```
        new Product("Laptop", "Electronics", 1200),
```

```
        new Product("Phone", "Electronics", 800),
```

```
        new Product("Shoes", "Fashion", 100),
```

```
        new Product("T-Shirt", "Fashion", 50),
```

```
        new Product("Fridge", "Appliances", 1500),
```

```
        new Product("Oven", "Appliances", 700)
```

```
    );
```

```
    Map<String, List<Product>> groupedByCategory = products.stream()
```

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



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

    Map<String, Optional<Product>> mostExpensiveByCategory =
products.stream()

    .collect(Collectors.groupingBy(product -> product.category,

        Collectors.maxBy(Comparator.comparingDouble(product ->
product.price))));

double averagePrice = products.stream()

    .mapToDouble(product -> product.price)

    .average()

    .orElse(0);

System.out.println("Products grouped by category:");

groupedByCategory.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.orElse(null)));

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

}

}
```



# DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

## 4. Output:

```
Products grouped by category:
Appliances: [Fridge - Category: Appliances, Price: 1500.0, Oven - Category: Appliances, Price: 700.0]
Fashion: [Shoes - Category: Fashion, Price: 100.0, T-Shirt - Category: Fashion, Price: 50.0]
Electronics: [Laptop - Category: Electronics, Price: 1200.0, Phone - Category: Electronics, Price: 800.0]

Most expensive product in each category:
Appliances: Fridge - Category: Appliances, Price: 1500.0
Fashion: Shoes - Category: Fashion, Price: 100.0
Electronics: Laptop - Category: Electronics, Price: 1200.0

Average price of all products: 725.0
PS C:\Users\harsh\OneDrive\Documents\Java Sem 6>
Ready
```

## 5. Learning Outcomes:

- Grouping Data.
- Using Java Streams.
- Handling race conditions in a multi-threaded environment.
- Taking user input for dynamic seat selection and priority assignment.
- Efficient Data Analysis.