1. Lambda Expressions – Case Study: Sorting and Filtering Employees

Scenario:

You are building a human resource management module. You need to:

• Sort employees by name or salary.

• Filter employees with a salary above a certain threshold.

Use Case:

Instead of creating multiple comparator classes or anonymous classes, you use Lambda expressions to sort and filter employee records in a concise and readable manner.

```
public class Employee {
    private String name;
    private double salary;
    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }
    public String getName() {
        return name;
    }
    public double getSalary() {
        return salary;
    }
    @Override
    public String toString() {
        return "Employee{name='" + name + "', salary=" + salary + "}";
    }
}
```

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

public class HRManagement {
    public static void main(String[] args) {

        List<Employee> employees = Arrays.asList(
            new Employee("Alice", 55000),
            new Employee("Bob", 48000),
            new Employee("Charlie", 70000),
            new Employee("David", 60000)
        );

        //  Sort by Name
        System.out.println("Sorted by Name:");
        employees.stream()
                .sorted((e1, e2) -> e1.getName().compareToIgnoreCase(e2.getName()))
                .forEach(System.out::println);

        // Sort by Salary (Ascending)
        System.out.println("\nSorted by Salary (Ascending):");
        employees.stream()
                .sorted(Comparator.comparingDouble(Employee::getSalary))
                .forEach(System.out::println);

        //  Sort by Salary (Descending)
        System.out.println("\nSorted by Salary (Descending):");
        employees.stream()
                .sorted((e1, e2) -> Double.compare(e2.getSalary(), e1.getSalary()))
                .forEach(System.out::println);
```

```
        // 🔹 Filter employees with salary > 55000
        double threshold = 55000;
        System.out.println("\nEmployees with salary > " + threshold + ":");
        employees.stream()
                .filter(emp -> emp.getSalary() > threshold)
                .forEach(System.out::println);
    }
}
```

2. Stream API & Operators – Case Study: Order Processing System

Scenario:

In an e-commerce application, you must:

• Filter orders above a certain value.

• Count total orders per customer.

• Sort and group orders by product category.

Use Case:

Streams help to process collections like orders using operators like filter, map, collect, sorted, and groupingBy to build readable pipelines for data processing.

```
public class Order {
    private String orderId;
    private String customerName;
    private String category;
    private double amount;

    public Order(String orderId, String customerName, String category, double amount) {
        this.orderId = orderId;
        this.customerName = customerName;
        this.category = category;
        this.amount = amount;
    }
```

```
    public String getOrderId() {
        return orderId;
    }
    public String getCustomerName() {
        return customerName;
    }
    public String getCategory() {
        return category;
    }
    public double getAmount() {
        return amount;
    }
    @Override
    public String toString() {
        return "Order{" +
                "orderId='" + orderId + '\'' +
                ", customerName='" + customerName + '\'' +
                ", category='" + category + '\'' +
                ", amount=" + amount +
                '}';
    }
}

import java.util.*;
import java.util.stream.Collectors;

public class OrderProcessing {
    public static void main(String[] args) {
```

```
List<Order> orders = Arrays.asList(
        new Order("O1", "Alice", "Electronics", 1200),
        new Order("O2", "Bob", "Clothing", 800),
        new Order("O3", "Alice", "Electronics", 500),
        new Order("O4", "Charlie", "Books", 300),
        new Order("O5", "Bob", "Books", 400),
        new Order("O6", "Alice", "Clothing", 700)
    );

    //  1. Filter orders above certain value (e.g., > 700)
    System.out.println("Orders above 700:");
    orders.stream()
        .filter(order -> order.getAmount() > 700)
        .forEach(System.out::println);

    // 2. Count total orders per customer
    System.out.println("\nTotal orders per customer:");
    Map<String, Long> ordersPerCustomer = orders.stream()
        .collect(Collectors.groupingBy(Order::getCustomerName, Collectors.counting()));
    ordersPerCustomer.forEach((customer, count) ->
        System.out.println(customer + ": " + count + " orders"));

    // 3. Sort by category and group orders by category
    System.out.println("\nOrders grouped by category (sorted within each group):");
    Map<String, List<Order>> groupedByCategory = orders.stream()
        .sorted(Comparator.comparing(Order::getAmount))
        .collect(Collectors.groupingBy(Order::getCategory, LinkedHashMap::new,
Collectors.toList()));
    groupedByCategory.forEach((category, orderList) -> {
        System.out.println("Category: " + category);
        orderList.forEach(System.out::println);        }); } }
```

Step 1: Create a Custom Functional Interface

```
@FunctionalInterface
public interface LogFilter {
   boolean shouldLog(String message);
}
```

Step 2: Logger Utility Class

```
import java.util.function.Consumer;
import java.util.function.Predicate;

public class LoggerUtil {
   // Using custom LogFilter
   public static void log(String message, LogFilter filter) {
      if (filter.shouldLog(message)) {
         System.out.println("[CustomLogFilter] " + message);
      }
   }
   // Using built-in Predicate
   public static void logWithPredicate(String message, Predicate<String> predicate) {
      if (predicate.test(message)) {
         System.out.println("[Predicate] " + message);
      }
   }
   // Using built-in Consumer
   public static void processLogs(String message, Consumer<String> consumer) {
      consumer.accept(message);
   }
}
```

Step 3: Main Class – Using Lambda with Functional Interfaces

```
public class LoggerApp {
    public static void main(String[] args) {

        String errorLog = "ERROR: Something went wrong!";
        String infoLog = "INFO: Application started.";
        String debugLog = "DEBUG: Debugging app.";

        // Use custom LogFilter (only log ERROR messages)
        LoggerUtil.log(errorLog, msg -> msg.startsWith("ERROR"));
        LoggerUtil.log(infoLog, msg -> msg.startsWith("ERROR")); // won't log

        // Use built-in Predicate (only log INFO or DEBUG)
        LoggerUtil.logWithPredicate(infoLog, msg -> msg.contains("INFO"));
        LoggerUtil.logWithPredicate(debugLog, msg -> msg.contains("DEBUG"));

        // Use built-in Consumer (custom action on message)
        LoggerUtil.processLogs(errorLog, msg -> System.out.println("[CONSUMER-LOG] -> " + msg));
        LoggerUtil.processLogs(infoLog, msg -> {
            if (msg.contains("INFO"))
                System.out.println("[INFO-LOG] -> " + msg);
        });
    }
}
```

4. Default Methods in Interfaces – Case Study: Payment Gateway Integration

Scenario:

You're integrating multiple payment methods (PayPal, UPI, Cards) using interfaces.

Use Case:

You use default methods in interfaces to provide shared logic (like transaction logging or currency conversion) without forcing each implementation to re-define them.

Step 1: Define the Interface with Default Methods

```
public interface PaymentGateway {

    void processPayment(double amount);

    //  Default method for transaction logging
    default void logTransaction(String method, double amount) {
        System.out.println("Logging transaction via " + method + ": $" + amount);
    }

    // Default method for currency conversion
    default double convertCurrency(double amount, String fromCurrency, String toCurrency) {
        // Simplified example: fixed conversion rate
        if (fromCurrency.equals("USD") && toCurrency.equals("INR")) {
            return amount * 83.0;
        }
        return amount; // Assume same currency if no match
    }
}
```

Step 2: Implement Different Payment Methods

```
public class PayPalPayment implements PaymentGateway {
    @Override
    public void processPayment(double amount) {
        double converted = convertCurrency(amount, "USD", "INR");
        System.out.println("Processing PayPal payment of ₹" + converted);
        logTransaction("PayPal", converted);
    }
}
```

```
public class UpiPayment implements PaymentGateway {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing UPI payment of ₹" + amount);
        logTransaction("UPI", amount);
    }
}
```

Use The Payment class in main

```
public class CardPayment implements PaymentGateway {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing Card payment of ₹" + amount);
        logTransaction("Card", amount);
    }
}

public class PaymentApp {
    public static void main(String[] args) {

        PaymentGateway paypal = new PayPalPayment();
        PaymentGateway upi = new UpiPayment();
        PaymentGateway card = new CardPayment();

        System.out.println("=== PayPal ===");
        paypal.processPayment(100);

        System.out.println("\n=== UPI ===");
        upi.processPayment(500);

        System.out.println("\n=== Card ===");
        card.processPayment(1000) } }
```