Java8 - Case Study

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.

Code:

```
package Task;

import java.util.Arrays;

import java.util.List;

public class HumanResourceManagement {

    static class Employee{

            private String name;

            private double salary;

            public Employee(String name, double salary) {

                    this.name = name;

                    this.salary = salary;

            }

            @Override

            public String toString() {

              return name + " -" + salary+"Rs";

            }

    }

    public static void main(String[] args) {

            List<Employee> empList = Arrays.asList(

                            new Employee("David",50000),

                            new Employee("Alice",40000),

                            new Employee("Bob",30000)

                            );

            //sort by  name

            empList.sort((e1, e2) -> e1.name.compareTo(e2.name));
```

```
        System.out.println("Sorted by Name:");
        empList.forEach(System.out::println);

        // Sort by salary using lambda
        empList.sort((e1, e2) -> Double.compare(e1.salary, e2.salary));
        System.out.println("\nSorted by Salary:");
        empList.forEach(System.out::println);


             //filter emp with salary >
              double threshold = 35000;
          System.out.println("\nEmployees with salary > ₹" + threshold + ":");
            for (Employee e : empList) {
                if (e.salary > threshold) {
                    System.out.println(e);
                }
            }
        }
    }
```

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.

Code:

```
package Task;

import java.util.Arrays;

import java.util.HashMap;

import java.util.List;

import java.util.Map;
import java.util.stream.Collectors;

    class Order {

      private String orderId;

      private String customerName;

      private String productCategory;

      private double amount;

      public Order(String orderId, String customerName, String productCategory, double amount) {

        this.orderId = orderId;

        this.customerName = customerName;

        this.productCategory = productCategory;

        this.amount = amount;

      }

            public String getOrderId() {

                    return orderId;

            }

            public String getCustomerName() {

                    return customerName;

            }

            public String getProductCategory() {

                    return productCategory;

            }

            public double getAmount() {

                    return amount;

            }


            @Override

      public String toString() {

        return "[" + orderId + "] " + customerName + " - " + productCategory + " = " + amount;

      }

}

public class ECommerceApplication {
    public static void main(String[] args) {
            List<Order> orders = Arrays.asList(
                new Order("O101", "Alice", "Electronics", 12000),
                new Order("O102", "Bob", "Books", 800),
                new Order("O103", "Alice", "Books", 500),
                new Order("O104", "David", "Clothing", 3000),
                new Order("O105", "Alice", "Electronics", 15000),
                new Order("O106", "Bob", "Clothing", 2200),
                new Order("O107", "David", "Books", 700)
              );
            // 1. Filter orders above ₹2000
        System.out.println("Orders above ₹2000:");
        for (Order o : orders) {
          if (o.getAmount() > 2000) {
            System.out.println(o);
          }
        }
        // 2. Count orders per customer
        System.out.println("\nTotal orders per customer:");
        Map<String, Integer> countMap = new HashMap<>();
        for (Order o : orders) {
          countMap.put(o.getCustomerName(), countMap.getOrDefault(o.getCustomerName(), 0) + 1);
        }
        for (String customer : countMap.keySet()) {
          System.out.println(customer + ": " + countMap.get(customer) + " orders");
        }
    }
}
```

3Functional Interfaces – Case Study: Custom Logger

Scenario:

You want to create a logging utility that allows:

- Logging messages conditionally.
- Reusing common log filtering logic.

Use Case: You define a custom LogFilter functional interface and allow users to pass behavior using lambdas. You also utilizee built-in interfaces like Predicate and Consumer.

Code:

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

import java.util.function.Consumer;

// 1. Custom functional interface

@FunctionalInterface

interface LogFilter {

   boolean shouldLog(String message);

}

// 2. Logger class

class Logger {

   private LogFilter filter;

   public Logger(LogFilter filter) {

      this.filter = filter;

   }

   public void log(String message) {

      if (filter.shouldLog(message)) {

         System.out.println("LOG: " + message);

      }

   }

}

public class CustomLoggerExample {

   public static void main(String[] args) {

      // Useing custom LogFilter with lambda to log only ERROR messages

      Logger errorLogger = new Logger(msg -> msg.contains("ERROR"));

      errorLogger.log("ERROR: Something went wrong");

      errorLogger.log("INFO: Server started"); // Won't log

      // Using Predicate to log WARNING messages

      Predicate<String> warningFilter = msg -> msg.contains("WARNING");

      String warnMsg = "WARNING: Low battery";

      if (warningFilter.test(warnMsg)) {
            System.out.println("WARN LOG: " + warnMsg);
        }
        // Using Consumer to log message
        Consumer<String> upperLogger = msg -> System.out.println(">> " + msg.toUpperCase());
        upperLogger.accept("this is a custom log message");
    }
}
```

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.

Code:

```
package Task;
interface Payment {
    void pay(double amount);
    // default method
    default void log(double amount) {
        System.out.println("Payment of " + amount + " was successful.");
    }
}
class PayPal implements Payment {
    public void pay(double amount) {
        System.out.println("Paid using PayPal");
        log(amount); // using default method
    }
}
class UPI implements Payment {
    public void pay(double amount) {
        System.out.println("Paid using UPI");
        log(amount);
    }
}
class Card implements Payment {
   public void pay(double amount) {
     System.out.println("Paid using Card");
      log(amount);
   }
}
//Main class
public class PaymentTest  {
   public static void main(String[] args) {
      Payment p1 = new PayPal();
      p1.pay(1000);
      Payment p2 = new UPI();
      p2.pay(500);
      Payment p3 = new Card();
      p3.pay(2000);
   }
}
```

5. Method References – Case Study: Notification System

Scenario: You're sending different types of notifications (Email, SMS, Push). The methods for sending are already defined in separate classes.

Use Case: You use method references (e.g., NotificationService::sendEmail) to refer to existing static or instance methods, making your event dispatcher concise and readable**.**

Code:

```
class NotificationService {
   public static void sendEmail(String msg) {
      System.out.println("Email sent: " + msg);
   }
   public static void sendSMS(String msg) {
      System.out.println("SMS sent: " + msg);
   }
   public void pushNotification(String msg) {
        System.out.println("Push notification: " + msg);
    }
}
public class NotificationSystem {
    public static void main(String[] args) {
        Runnable email = () -> NotificationService.sendEmail("Welcome!");
        Runnable sms = () -> NotificationService.sendSMS("OTP: 1234");
        email.run();
        sms.run();
    }
}
```

6. Optional Class – Case Study: User Profile Management

Code:

```
import java.util.Optional;
class UserProfile {
    private Optional<String> email;
    public UserProfile(String email) {
        this.email = Optional.ofNullable(email);
    }
    public void showEmail() {
        email.ifPresentOrElse(e -> System.out.println("Email: " + e),
            () -> System.out.println("Email not provided")
        );
    }
}
public class UserProfileExample {
    public static void main(String[] args) {
        UserProfile u1 = new UserProfile("abc@example.com");
        u1.showEmail();    }
}
```

7.Date and Time API – Case Study: Booking System

Code:

```
import java.time.*;


public class BookingSystem {
   public static void main(String[] args) {
      LocalDate checkIn = LocalDate.of(2024, 12, 10);
      LocalDate checkOut = LocalDate.of(2024, 12, 15);
      // Validate dates
      if (checkOut.isAfter(checkIn)) {
         Period stayDuration = Period.between(checkIn, checkOut);
         System.out.println("Stay Duration: " + stayDuration.getDays() + " days");
      } else {
         System.out.println("Check-out date must be after check-in date.");
      }
    }
}
```

8. ExecutorService – Case Study: File Upload Service

Code:

```
import java.util.concurrent.*;


public class FileUploadService {
   public static void main(String[] args) {
      ExecutorService executor = Executors.newFixedThreadPool(3);


      Runnable upload1 = () -> System.out.println("Uploading file1...");
      Runnable upload2 = () -> System.out.println("Uploading file2...");
      Runnable upload3 = () -> System.out.println("Uploading file3...");
      executor.submit(upload1);
      executor.submit(upload2);
```

```
        executor.submit(upload3);


        executor.shutdown(); // shutdown
    }
}
```