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.

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

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 name + " - ₹" + salary;
    }
}

public class EmployeeManagement {
        public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(
            new Employee("Ishant", 55000),
            new Employee("Manavi", 48000),
            new Employee("Akash", 72000),
            new Employee("Priya", 39000)
        );

        System.out.println(" Original List:");
```

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


        List<Employee> sortedByName = new ArrayList<>(employees);
        sortedByName.sort((e1, e2) -> e1.getName().compareToIgnoreCase(e2.getName()));
        System.out.println("\n Sorted by Name:");
        sortedByName.forEach(System.out::println);


        List<Employee> sortedBySalary = new ArrayList<>(employees);
        sortedBySalary.sort((e1, e2) -> Double.compare(e1.getSalary(), e2.getSalary()));
        System.out.println("\n Sorted by Salary:");
        sortedBySalary.forEach(System.out::println);


        List<Employee> highEarners = employees.stream()
                                    .filter(e -> e.getSalary() > 50000)
                                    .collect(Collectors.toList());
        System.out.println("\n Employees with Salary > ₹50,000:");
        highEarners.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.

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

class Order {
    private String orderId;
    private String customerName;
    private String productCategory;
    private double orderValue;

    public Order(String orderId, String customerName, String productCategory, double orderValue)
```

```
{
        this.orderId = orderId;
        this.customerName = customerName;
        this.productCategory = productCategory;
        this.orderValue = orderValue;
    }

    public String getOrderId() {
        return orderId;
    }

    public String getCustomerName() {
        return customerName;
    }

    public String getProductCategory() {
        return productCategory;
    }

    public double getOrderValue() {
        return orderValue;
    }

    @Override
    public String toString() {
        return orderId + " | " + customerName + " | " + productCategory + " | ₹" + orderValue;
    }
}

public class OrderProcessing {
    public static void main(String[] args) {
        List<Order> orders = Arrays.asList(
            new Order("O101", "Ishant", "Electronics", 2500),
            new Order("O102", "Manavi", "Books", 500),
            new Order("O103", "Ishant", "Clothing", 1200),
            new Order("O104", "Akash", "Electronics", 800),
            new Order("O105", "Manavi", "Clothing", 1500),
            new Order("O106", "Priya", "Books", 2000),
            new Order("O107", "Priya", "Electronics", 3000)
        );



        System.out.println(" Orders above ₹1000:");
        orders.stream()
              .filter(order -> order.getOrderValue() > 1000)
              .forEach(System.out::println);
```

```
        System.out.println("\n Total Orders per Customer:");
        Map<String, Long> ordersPerCustomer = orders.stream()
            .collect(groupingBy(Order::getCustomerName, counting()));
        ordersPerCustomer.forEach((customer, count) ->
            System.out.println(customer + ": " + count + " orders"));


        System.out.println("\n Orders Grouped by Product Category (Sorted by Value):");
        Map<String, List<Order>> ordersByCategory = orders.stream()
            .sorted(Comparator.comparing(Order::getOrderValue).reversed())
            .collect(groupingBy(Order::getProductCategory));

        ordersByCategory.forEach((category, orderList) -> {
            System.out.println("\n " + category + ":");
            orderList.forEach(System.out::println);
        });
    }
}
```

3. Functional 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 utilize built-in interfaces like `Predicate` and `Consumer`.

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


@FunctionalInterface
interface LogFilter {
    boolean shouldLog(String level, String message);
}

class Logger {
    private LogFilter logFilter;
```

```
    public Logger(LogFilter logFilter) {
        this.logFilter = logFilter;
    }

    public void log(String level, String message) {
        if (logFilter.shouldLog(level, message)) {
            System.out.println("[" + level.toUpperCase() + "] " + message);
        }
    }


    public void log(String message, Predicate<String> predicate, Consumer<String> action) {
        if (predicate.test(message)) {
            action.accept(message);
        }
    }
}

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

        Logger errorLogger = new Logger((level, msg) -> level.equalsIgnoreCase("error"));
        Logger debugLogger = new Logger((level, msg) -> level.equalsIgnoreCase("debug"));

        errorLogger.log("error", "Null pointer exception occurred!");
        errorLogger.log("info", "Just an info");

        debugLogger.log("debug", "Debugging login issue...");
        debugLogger.log("error", "Should not be printed by debugLogger");


        Logger flexibleLogger = new Logger((lvl, msg) -> true);

        Predicate<String> startsWithLogin = msg -> msg.startsWith("Login");
        Consumer<String> printToConsole = msg -> System.out.println("[CUSTOM] " + msg);

        flexibleLogger.log("Login successful for user: ishant", startsWithLogin, printToConsole);
        flexibleLogger.log("Signup success", startsWithLogin, printToConsole);
    }
}
```

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

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

```
interface PaymentGateway {
    void processPayment(double amount);

    default void logTransaction(String method, double amount) {
        System.out.println("[LOG] Payment of ₹" + amount + " processed via " + method);
    }

    default double convertCurrency(double usdAmount, double rate) {
        return usdAmount * rate;
    }
}

class PayPalGateway implements PaymentGateway {
    public void processPayment(double amount) {
        double inrAmount = convertCurrency(amount, 83.2);
        System.out.println("Processing PayPal payment: ₹" + inrAmount);
        logTransaction("PayPal", inrAmount);
    }
}

class UpiGateway implements PaymentGateway {
    public void processPayment(double amount) {
        System.out.println("Processing UPI payment: ₹" + amount);
        logTransaction("UPI", amount);
    }
}

class CardGateway implements PaymentGateway {
    public void processPayment(double amount) {
        System.out.println("Processing Card payment: ₹" + amount);
        logTransaction("Card", amount);
    }
}

public class PaymentDemo {
    public static void main(String[] args) {
        PaymentGateway paypal = new PayPalGateway();
        PaymentGateway upi = new UpiGateway();
        PaymentGateway card = new CardGateway();

        paypal.processPayment(100);
        upi.processPayment(2500);
        card.processPayment(1500);
    }
}
```

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.

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.

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


class EmailService {
    public static void sendEmail(String message) {
        System.out.println(" Email Sent: " + message);
    }
}

class SmsService {
    public void sendSms(String message) {
        System.out.println(" SMS Sent: " + message);
    }
}

class PushService {
    public void sendPush(String message) {
        System.out.println(" Push Notification Sent: " + message);
    }
}


class NotificationDispatcher {
    public void dispatch(String message, Consumer<String> methodRef) {
        methodRef.accept(message);
    }
}

public class NotificationDemo {

        public static void main(String[] args) {

                NotificationDispatcher dispatcher = new NotificationDispatcher();

        SmsService smsService = new SmsService();
```

```
        PushService pushService = new PushService();


        dispatcher.dispatch("Welcome to the system!", EmailService::sendEmail);
        dispatcher.dispatch("Your OTP is 123456", smsService::sendSms);
        dispatcher.dispatch("You have a new alert", pushService::sendPush);
    }

     }
```

6. Optional Class – Case Study: User Profile Management

Scenario:
User details like email or phone number may be optional during registration.

Use Case:
To avoid `NullPointerException`, you wrap potentially null fields in `Optional`. This forces developers to handle absence explicitly using methods like `orElse`, `ifPresent`, or `map`.

```
import java.util.Optional;


class UserProfile {

    private String username;

    private Optional<String> email;

    private Optional<String> phoneNumber;


    // Constructor

    public UserProfile(String username, String email, String phoneNumber) {

        this.username = username;

        this.email = Optional.ofNullable(email);

        this.phoneNumber = Optional.ofNullable(phoneNumber);

    }


    public String getUsername() {

        return username;

    }
```

```
    public Optional<String> getEmail() {
        return email;
    }

    public Optional<String> getPhoneNumber() {
        return phoneNumber;
    }
}

public class UserProfileDemo {
        public static void main(String[] args) {
        UserProfile user1 = new UserProfile("ishant", "ishant@example.com", null);
        UserProfile user2 = new UserProfile("manavi", null, "1212212");

        printUserInfo(user1);
        System.out.println("--------------");
        printUserInfo(user2);
    }

    public static void printUserInfo(UserProfile user) {
        System.out.println(" Username: " + user.getUsername());


        user.getEmail().ifPresent(email ->
            System.out.println("Email: " + email)
        );


        String phone = user.getPhoneNumber().orElse("No phone provided");
```

```
        System.out.println("Phone: " + phone);



        String maskedEmail = user.getEmail()
                        .map(e -> e.replaceAll("(?<=.).(?=[^@]*?@)", "*"))
                        .orElse("Email not provided");
        System.out.println("Masked Email: " + maskedEmail);
    }


}
```

7. Date and Time API (java.time) – Case Study: Booking System

Scenario:
A hotel or travel booking system that:

- Calculates stay duration.
- Validates check-in/check-out dates.
- Schedules recurring events.

Use Case:
You use the new `LocalDate`, `LocalDateTime`, `Period`, and `Duration` classes to perform safe and readable date/time calculations.

```
import java.time.*;

import java.time.temporal.ChronoUnit;
import java.util.stream.IntStream;

class Booking {
    private String customerName;
    private LocalDate checkInDate;
    private LocalDate checkOutDate;

    public Booking(String customerName, LocalDate checkInDate, LocalDate checkOutDate) {
        if (checkOutDate.isBefore(checkInDate)) {
            throw new IllegalArgumentException(" Check-out date cannot be before check-in date.");
        }
        this.customerName = customerName;
        this.checkInDate = checkInDate;
        this.checkOutDate = checkOutDate;
```

```
    }

    public void printBookingSummary() {
        System.out.println(" Customer: " + customerName);
        System.out.println(" Check-In: " + checkInDate);
        System.out.println(" Check-Out: " + checkOutDate);

        Period stayPeriod = Period.between(checkInDate, checkOutDate);
        long nights = ChronoUnit.DAYS.between(checkInDate, checkOutDate);

        System.out.println(" Stay Duration: " + nights + " nights (" + stayPeriod.getDays() + " days)");
    }

    public void scheduleDailyRoomCleaning() {
        System.out.println(" Scheduled Room Cleaning:");
        IntStream.range(0, (int) ChronoUnit.DAYS.between(checkInDate, checkOutDate))
                .mapToObj(i -> checkInDate.plusDays(i))
                .forEach(date -> System.out.println("- " + date + " at 10:00 AM"));
    }
}

public class BookingSystemDemo {
        public static void main(String[] args) {
        LocalDate checkIn = LocalDate.of(2025, 7, 27);
        LocalDate checkOut = LocalDate.of(2025, 7, 30);

        Booking booking = new Booking("Ishant Singh", checkIn, checkOut);
        booking.printBookingSummary();
        System.out.println("-------------------------");
        booking.scheduleDailyRoomCleaning();


        System.out.println("\n Weekly Shuttle Schedule:");
        LocalDate shuttleStart = LocalDate.of(2025, 7, 27);
        for (int i = 0; i < 4; i++) {
            LocalDate nextPickup = shuttleStart.plusWeeks(i);
            System.out.println("Shuttle Pickup: " + nextPickup + " at 9:00 AM");
        }


        LocalDateTime now = LocalDateTime.now();
        LocalDateTime meeting = now.plusHours(3);
        Duration duration = Duration.between(now, meeting);
        System.out.println("\n Meeting in: " + duration.toHours() + " hours");
    }

}
```

8. Executor Service – Case Study: File Upload Service

Scenario:
You allow users to upload multiple files simultaneously and want to manage the processing efficiently.

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

class FileUploadTask implements Runnable {
    private String fileName;

    public FileUploadTask(String fileName) {
        this.fileName = fileName;
    }

    @Override
    public void run() {
        System.out.println(" Uploading " + fileName + " on thread: " +
Thread.currentThread().getName());
        try {
            Thread.sleep(2000);
            System.out.println(" Upload completed: " + fileName);
        } catch (InterruptedException e) {
            System.out.println("Upload interrupted: " + fileName);
        }
    }
}

public class FileUploadService {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);
        String[] files = { "doc1.pdf", "photo.jpg", "report.xlsx", "video.mp4", "slide.pptx" };

        for (String file : files) {
            executor.submit(new FileUploadTask(file));
        }

        executor.shutdown();
    }
}
```

Use Case:

You use `ExecutorService` to handle concurrent uploads by creating a thread pool, managing background tasks without blocking the UI or main thread.