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.*;


class Employee {
    String name;
    double salary;


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


    public String toString() {
        return name + " - ₹" + salary;
    }
}


public class LambdaExample {
    public static void main(String[] args) {
        List<Employee> list = Arrays.asList(
            new Employee("Raj", 55000),
            new Employee("Anita", 70000),
            new Employee("Vikram", 45000)
        );


        // Sort by name
        list.sort((e1, e2) -> e1.name.compareTo(e2.name));
```

```
        System.out.println("Sorted by name: " + list);

        // Filter salary > 50000
        list.stream()
            .filter(e -> e.salary > 50000)
            .forEach(e -> System.out.println("High Earner: " + 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.

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

class Order {
    String customer;
    String category;
    double value;

    Order(String customer, String category, double value) {
        this.customer = customer;
        this.category = category;
        this.value = value;
    }

    public String toString() {
        return customer + ": " + category + " - ₹" + value;
    }
}
```

```
public class StreamExample {
    public static void main(String[] args) {
        List<Order> orders = Arrays.asList(
            new Order("Amit", "Electronics", 1200),
            new Order("Amit", "Books", 300),
            new Order("Sara", "Electronics", 900),
            new Order("Sara", "Clothing", 1500)
        );

        // Filter orders > 1000
        orders.stream()
            .filter(o -> o.value > 1000)
            .forEach(System.out::println);

        // Count orders per customer
        Map<String, Long> orderCount = orders.stream()
            .collect(groupingBy(o -> o.customer, counting()));
        System.out.println("Orders per customer: " + orderCount);

        // Group by category
        Map<String, List<Order>> grouped = orders.stream()
            .collect(groupingBy(o -> o.category));
        System.out.println("Grouped by category: " + grouped);
    }
}
```

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.

```
@FunctionalInterface
```

```
interface LogFilter {
    boolean shouldLog(String msg);
}

public class Logger {
    public static void log(String msg, LogFilter filter) {
        if (filter.shouldLog(msg)) {
            System.out.println("LOG: " + msg);
        }
    }

    public static void main(String[] args) {
        log("ERROR: Disk full", msg -> msg.contains("ERROR"));
        log("INFO: App started", msg -> msg.contains("ERROR")); // won't log
    }
}
```

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.

```
interface Payment {
    void pay(double amount);

    default void logTransaction(double amount) {
        System.out.println("Transaction of ₹" + amount + " logged.");
    }
}

class PayPal implements Payment {
    public void pay(double amount) {
        System.out.println("Paid ₹" + amount + " via PayPal");
```

```
        logTransaction(amount);
    }
}
```

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.

```
class NotificationService {
    static void sendEmail(String msg) {
        System.out.println("Email sent: " + msg);
    }

    void sendSMS(String msg) {
        System.out.println("SMS sent: " + msg);
    }
}

public class MethodRefDemo {
    public static void main(String[] args) {
        Consumer<String> emailNotifier = NotificationService::sendEmail;
        emailNotifier.accept("Order confirmed!");

        NotificationService service = new NotificationService();
        Consumer<String> smsNotifier = service::sendSMS;
        smsNotifier.accept("OTP: 123456");
    }
}
```

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 User {
    Optional<String> email = Optional.ofNullable(null);
}

public class OptionalExample {
    public static void main(String[] args) {
        User user = new User();
        System.out.println(user.email.orElse("Email not provided"));

        user.email = Optional.of("user@example.com");
        user.email.ifPresent(e -> System.out.println("Email: " + e));
    }
}
```

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.*;

public class DateTimeDemo {
    public static void main(String[] args) {
        LocalDate checkIn = LocalDate.of(2025, 7, 24);
        LocalDate checkOut = LocalDate.of(2025, 7, 28);

        Period stayDuration = Period.between(checkIn, checkOut);
```

```
        System.out.println("Stay: " + stayDuration.getDays() + " days");

        LocalDateTime now = LocalDateTime.now();
        System.out.println("Current time: " + now);
    }
}
```

8. Executor Service – Case Study: File Upload Service Scenario: You allow users to upload multiple files simultaneously and want to manage the processing efficiently. Use Case: You use ExecutorService to handle concurrent uploads by creating a thread pool, managing background tasks without blocking the UI or main thread.

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

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

        Runnable uploadTask = () -> {
            System.out.println("Uploading file by: " + Thread.currentThread().getName());
        };

        for (int i = 0; i < 5; i++) {
            executor.submit(uploadTask);
        }

        executor.shutdown();
    }
}
```