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

```java
package Lambda_expression;

public class Employee {
   private String name;
   private double salary;

   // Constructor, getters, setters

   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;
   }
}
```

```java
package Lambda_expression;

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

public class LambdaEmployeeDemo {
   public static void main(String[] args) {
      List<Employee> employees = Arrays.asList(
         new Employee("Alice", 70000),
         new Employee("Bob", 45000),
         new Employee("Charlie", 55000)
      );

      // Sort by name
      employees.sort(Comparator.comparing(Employee::getName));
      System.out.println("Sorted by name: " + employees);
```

```java
        // Sort by salary
        employees.sort(Comparator.comparingDouble(Employee::getSalary));
        System.out.println("Sorted by salary: " + employees);

        // Filter by salary threshold
        double threshold = 50000;
        List<Employee> filtered = employees.stream()
            .filter(e -> e.getSalary() > threshold)
            .collect(Collectors.toList());
        System.out.println("Salary > " + threshold + ": " + filtered);
    }
}
```

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.

```java
package streamapi_operators;

public class order {
    private String customerName;
    private String productCategory;
    private double orderValue;

    public order(String customerName, String productCategory, double orderValue) {
        this.customerName = customerName;
        this.productCategory = productCategory;
        this.orderValue = orderValue;
    }

    public String getCustomerName() {
        return customerName;
    }

    public String getProductCategory() {
        return productCategory;
    }

    public double getOrderValue() {
        return orderValue;
    }

    @Override
```

```java
    public String toString() {
        return customerName + " - " + productCategory + ": $" + orderValue;
    }
}


package streamapi_operators;

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

public class StreamOrderDemo {
    public static void main(String[] args) {
        // Sample data
        List<order> orders = Arrays.asList(
            new order("Alice", "Electronics", 150),
            new order("Bob", "Books", 80),
            new order("Alice", "Books", 120),
            new order("Dave", "Electronics", 90),
            new order("Bob", "Electronics", 200)
        );

        // Filter Orders above a threshold
        double threshold = 100;
        List<order> highValueOrders = orders.stream()
            .filter(o -> o.getOrderValue() > threshold)
            .collect(Collectors.toList());
        System.out.println("High value orders: " + highValueOrders);

        // Count total orders per customer
        Map<String, Long> ordersPerCustomer = orders.stream()
            .collect(Collectors.groupingBy(order::getCustomerName, Collectors.counting()));
        System.out.println("Orders per customer: " + ordersPerCustomer);

        // Sort and group orders by product category
        Map<String, List<order>> ordersByCategory = orders.stream()
            .sorted(Comparator.comparingDouble(order::getOrderValue).reversed())
            .collect(Collectors.groupingBy(order::getProductCategory));
        System.out.println("Sorted & grouped by category:");
        ordersByCategory.forEach((category, orderList) -> {
            System.out.println(category + ": " + orderList);
        });
    }
}
```

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.

```java
package logging_message;

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

```java
package logging_message;

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

// Import your LogFilter and LoggerUtil classes depending on your package structure
import logging_message.LogFilter;
import logging_message.LoggerUtil;


public class LoggerDemo {

    public static void main(String[] args) {
        // Common logger: print to console
        Consumer<String> consoleLogger = msg -> System.out.println("[LOG] " + msg);

        // Custom LogFilter implementation using lambda (e.g., log messages containing
"ERROR")
        LogFilter errorFilter = msg -> msg.contains("ERROR");

        // Using custom LogFilter with LoggerUtil
        LoggerUtil.log("This is an ERROR message", errorFilter, consoleLogger);
        LoggerUtil.log("This is an INFO message", errorFilter, consoleLogger);

        // Predicate built-in interface for filtering (log messages longer than 20 chars)
        Predicate<String> longMessageFilter = msg -> msg.length() > 20;

        // Using Predicate with LoggerUtil
        LoggerUtil.log("Short msg", longMessageFilter, consoleLogger);
        LoggerUtil.log("This is a much longer informational message", longMessageFilter,
consoleLogger);
```

```
        // Reusing common filters with different logger consumers (e.g., console or file logger)
        Consumer<String> fileLogger = msg -> {
            // Simulated file logging (for this demo just print prefixed)
            System.out.println("[FILE LOG] " + msg);
        };

        LoggerUtil.log("ERROR: Disk space is low", errorFilter, fileLogger);
    }
}
```

```
package logging_message;

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

public class LoggerUtil {

    // Logs using custom LogFilter and logs messages that satisfy the condition
    public static void log(String message, LogFilter filter, Consumer<String> logger) {
        if (filter.shouldLog(message)) {
            logger.accept(message);
        }
    }

    // Logs using Predicate (built-in filter) and Consumer (logger)
    public static void log(String message, Predicate<String> filter, Consumer<String> logger)
{
        if (filter.test(message)) {
            logger.accept(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.

```
package payment_methods;

public class CardPayment implements Payment {
```

```java
    @Override
    public void pay(double amount) {
        // For card, suppose we have a conversion too
        double amountInINR = convertCurrency(amount, 82.5);
        System.out.println("Processing Card payment of INR " + amountInINR);

        // Log transaction
        logTransaction("Card", amountInINR);
    }
}
```

```java
package payment_methods;

public interface Payment {

    // Each payment method must implement this to perform the actual payment
    void pay(double amount);

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

    // Default method for currency conversion (e.g., from USD to INR)
    default double convertCurrency(double amount, double conversionRate) {
        double converted = amount * conversionRate;
        System.out.println("Converted amount: " + converted);
        return converted;
    }
}
```

```java
package payment_methods;

public class PaymentDemo {
    public static void main(String[] args) {
        Payment paypal = new PayPalPayment();
        Payment upi = new UPIPayment();
        Payment card = new CardPayment();

        double amountUSD = 100;

        paypal.pay(amountUSD);   // PayPal with conversion & logging
        upi.pay(7500);           // UPI payment without conversion
        card.pay(amountUSD);     // Card payment with conversion & logging
    }
}
```

```java
package payment_methods;

public class PayPalPayment implements Payment {

    @Override
    public void pay(double amount) {
        // Convert currency if needed (e.g., USD to INR)
        double amountInINR = convertCurrency(amount, 82.5);
        // Process PayPal-specific payment with converted amount
        System.out.println("Processing PayPal payment of INR " + amountInINR);

        // Log transaction
        logTransaction("PayPal", amountInINR);
    }
}
```

```java
package payment_methods;

public class UPIPayment implements Payment {

    @Override
    public void pay(double amount) {
        // UPI transfer is usually in local currency; no conversion needed in this example
        System.out.println("Processing UPI payment of INR " + amount);

        // Log transaction
        logTransaction("UPI", 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.

```java
package Notification_methods;

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

```
package Notification_methods;

public class NotificationDemo {
    public static void main(String[] args) {
        NotificationDispatcher dispatcher = new NotificationDispatcher();

        dispatcher.dispatch("EMAIL", "Welcome, user!");
        dispatcher.dispatch("SMS", "Your OTP is 123456");
        dispatcher.dispatch("PUSH", "New feature available!");
        dispatcher.dispatch("CALL", "This type is unknown"); // Will print 'Unknown
notification type'
    }
}
```

```
package Notification_methods;

import java.util.HashMap;
import java.util.Map;

public class NotificationDispatcher {
    private Map<String, Notifier> notificationMap = new HashMap<>();

    public NotificationDispatcher() {
        // Register method references for each notification type
        notificationMap.put("EMAIL", EmailService::sendEmail);
        notificationMap.put("SMS", SMSService::sendSMS);
        notificationMap.put("PUSH", PushService::sendPush);
    }

    public void dispatch(String type, String message) {
        Notifier notifier = notificationMap.get(type);
        if (notifier != null) {
            notifier.notify(message);
        } else {
            System.out.println("Unknown notification type: " + type);
        }
    }
}
```

```
package Notification_methods;

@FunctionalInterface
public interface Notifier {
    void notify(String message);
}
```

```
package Notification_methods;
```

```java
public class PushService {
        public static void sendPush(String message) {
        System.out.println("Sending PUSH: " + message);
    }

}
```

```java
package Notification_methods;

public class SMSService {
        public static void sendSMS(String message) {
        System.out.println("Sending SMS: " + message);
    }

}
```

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

```java
package email_phone;


public class RegistrationDemo {
    public static void main(String[] args) {
        User user1 = new User("alice", null, "9876543210");
        User user2 = new User("bob", "bob@example.com", null);

        // Print email if present
        user1.getEmail().ifPresent(email -> System.out.println("Alice's email: " + email));
        user2.getEmail().ifPresent(email -> System.out.println("Bob's email: " + email));

        // Provide a default phone number if missing
        String phone = user2.getPhoneNumber().orElse("No phone provided");
        System.out.println("Bob's phone: " + phone);

        // Safely get email domain or default string
        String emailDomain = user1.getEmail()
                .map(email -> email.substring(email.indexOf('@') + 1))
                .orElse("No email domain");
        System.out.println("Alice's email domain: " + emailDomain);
```

```java
    }
}
```

```java
package email_phone;

import java.util.Optional;

public class User {
    private String username;
    private Optional<String> email;
    private Optional<String> phoneNumber;

    public User(String username, String email, String phoneNumber) {
        this.username = username;
        this.email = Optional.ofNullable(email);          // Wraps value or null
        this.phoneNumber = Optional.ofNullable(phoneNumber);
    }

    public String getUsername() { return username; }
    public Optional<String> getEmail() { return email; }
    public Optional<String> getPhoneNumber() { return phoneNumber; }
}
```

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.

```java
package DateTime;

import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class Booking {
    private LocalDate checkInDate;
    private LocalDate checkOutDate;
    private String guestName;

    public Booking(String guestName, LocalDate checkInDate, LocalDate checkOutDate) {
        if (!isValidDates(checkInDate, checkOutDate)) {
```

```java
            throw new IllegalArgumentException("Check-out date must be after check-in date");
        }
        this.guestName = guestName;
        this.checkInDate = checkInDate;
        this.checkOutDate = checkOutDate;
    }

    // Validate check-in is before check-out
    public static boolean isValidDates(LocalDate checkIn, LocalDate checkOut) {
        return checkIn != null && checkOut != null && checkIn.isBefore(checkOut);
    }

    // Calculate stay duration in days
    public long getStayDuration() {
        return ChronoUnit.DAYS.between(checkInDate, checkOutDate);
    }

    // Getters
    public LocalDate getCheckInDate() { return checkInDate; }
    public LocalDate getCheckOutDate() { return checkOutDate; }
    public String getGuestName() { return guestName; }

    @Override
    public String toString() {
        return String.format("Booking for %s from %s to %s (%d nights)",
                guestName, checkInDate, checkOutDate, getStayDuration());
    }
}
```

```java
package DateTime;

import java.time.LocalDate;
import java.time.DayOfWeek;
import java.util.List;

public class BookingSystemDemo {
    public static void main(String[] args) {
        // Valid booking example
        LocalDate checkIn = LocalDate.of(2025, 8, 1);
        LocalDate checkOut = LocalDate.of(2025, 8, 5);

        if (!Booking.isValidDates(checkIn, checkOut)) {
            System.out.println("Invalid booking dates!");
            return;
        }

        Booking booking = new Booking("John Doe", checkIn, checkOut);
```

```
        System.out.println(booking);
        System.out.println("Stay duration (nights): " + booking.getStayDuration());

        // Invalid booking (check-out before check-in) - throws exception
        try {
            Booking invalidBooking = new Booking("Jane Smith", LocalDate.of(2025, 8, 10),
LocalDate.of(2025, 8, 8));
        } catch (IllegalArgumentException e) {
            System.out.println("Caught error for invalid booking dates: " + e.getMessage());
        }

        // Schedule a recurring event: every Monday from today for next 4 occurrences
        LocalDate startDate = LocalDate.now();
        List<LocalDate> recurringMondays =
RecurringEventScheduler.getRecurringDates(startDate, DayOfWeek.MONDAY, 4);
        System.out.println("Next 4 recurring Mondays:");
        recurringMondays.forEach(date -> System.out.println(date));
    }
}
package DateTime;

import java.time.DayOfWeek;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

public class RecurringEventScheduler {

    /**
     * Gets the next N recurring dates from a start date based on a day of week.
     * For example, every Monday starting from a given date.
     */
    public static List<LocalDate> getRecurringDates(LocalDate startDate, DayOfWeek
dayOfWeek, int occurrences) {
        List<LocalDate> dates = new ArrayList<>();
        LocalDate nextDate = startDate;

        // Move to the first dayOfWeek if startDate is not that day
        while (nextDate.getDayOfWeek() != dayOfWeek) {
            nextDate = nextDate.plusDays(1);
        }

        for (int i = 0; i < occurrences; i++) {
            dates.add(nextDate);
            nextDate = nextDate.plusWeeks(1); // weekly recurrence
        }
        return dates;
    }
}
```

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

```java
package FileUpload;

import java.io.File;
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class FileUploadManager {
   public static void main(String[] args) {
      // 1. Prepare files (replace with actual file selection)
      List<File> filesToUpload = Arrays.asList(
            new File("file1.txt"),
            new File("file2.txt"),
            new File("file3.txt"),
            new File("file4.txt")
      );

      // 2. Create a fixed thread pool (e.g., 3 concurrent uploads at most)
      ExecutorService executor = Executors.newFixedThreadPool(3);

      // 3. Submit each file upload as a separate task
      for (File file : filesToUpload) {
         executor.submit(new FileUploadTask(file));
      }

      // 4. Optionally, shut down the executor once all tasks are submitted
      executor.shutdown();
   }
}
```

```java
package FileUpload;

import java.io.File;

public class FileUploadTask implements Runnable {
   private File file;
```

```
    public FileUploadTask(File file) {
        this.file = file;
    }

    @Override
    public void run() {
        // Simulate file upload; replace with real upload code
        System.out.println("Uploading: " + file.getName());
        try {
            Thread.sleep(2000); // Simulate upload time
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        System.out.println("Finished: " + file.getName());
    }
}
```