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.

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

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

    // Getters
    public String getName() { return name; }
public double getSalary() { return salary; }

    @Override
    public String toString() {
        return "Employee{name='" + name + "', salary=" + salary + "}";
    }
}
```

// Sorting employees by name or salary

```
import

java.util.*;

public class EmployeeSorter {
    public static void main(String[] args) {
List<Employee> employees = Arrays.asList(
new Employee("Alice", 70000),             new
Employee("Bob", 50000),            new
Employee("Charlie", 60000)
        );

        // Sort by name
        employees.sort((e1, e2) -> e1.getName().compareTo(e2.getName()));
System.out.println("Sorted by name:");
employees.forEach(System.out::println);

        // Sort by salary
        employees.sort((e1, e2) -> Double.compare(e1.getSalary(),
e2.getSalary()));
        System.out.println("\nSorted by salary:");
employees.forEach(System.out::println);
    }
}
```

// Filter employees with a salary above a certain threshold

2. Stream API & Operators – Case Study: Order `import java.util.stream.Collectors;`

```
public class EmployeeFilter {
    public static void main(String[] args) {
List<Employee> employees = Arrays.asList(
new Employee("Alice", 70000),             new
Employee("Bob", 50000),           new
Employee("Charlie", 60000)
        );
        double threshold = 55000;

        List<Employee> filtered = employees.stream()
            .filter(e -> e.getSalary() > threshold)
            .collect(Collectors.toList());

        System.out.println("Employees with salary above " + threshold + ":");
filtered.forEach(System.out::println);
    }
}
```

2. Stream API & Operators – Case Study: Order Processing System Scenario: In an ecommerce 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.

```
class Order {
String customer;
String category;
double amount;

    // Constructor, getters, toString
}

List<Order> orders = List.of(
    new Order("Alice", "Electronics", 120.0),
new Order("Bob", "Books", 80.0),     new
Order("Alice", "Books", 200.0),     new
Order("Charlie", "Electronics", 300.0)
);

// Filter orders above 100
List<Order> highValueOrders = orders.stream()
    .filter(o -> o.getAmount() > 100)
    .collect(Collectors.toList());
```

```
// Count orders per customer
Map<String, Long> ordersPerCustomer = orders.stream()
.collect(Collectors.groupingBy(Order::getCustomer, Collectors.counting()));
//Sort and group by category
Map<String, List<Order>> groupedByCategory = orders.stream()
 .sorted(Comparator.comparing(Order::getAmount).reversed())
 .collect(Collectors.groupingBy(Order::getCategory));
```

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

// Usage
Logger.log("ERROR: Disk full", msg -> msg.startsWith("ERROR"));
Or using built-in Predicate<String> :
Logger.log("INFO: Startup", ((Predicate<String>) msg ->
msg.contains("INFO"))::test);
```

4. Default Methods in Interfaces – Case Study: Payment Gateway Integration Scenario: You're integrating multiple payment methods (PayPal, UPI, Cards) using interfaces.

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

    default void logTransaction(double amount) {
        System.out.println("Transaction of $" + amount + " processed.");
    }
}  class PayPal implements PaymentMethod
{     public void pay(double amount) {
logTransaction(amount);
        System.out.println("Paid via PayPal");
    }
}
```

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 {
    public static void sendEmail(String msg) {
        System.out.println("Email: " + msg);
    }      public static void sendSMS(String
msg) {
        System.out.println("SMS: " + msg);
    }
}

List<Consumer<String>> notifiers = List.of(
    NotificationService::sendEmail,
    NotificationService::sendSMS
);
notifiers.forEach(n -> n.accept("Your order has shipped!"));
```

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.

```
class User {
    private Optional<String> email;

    public User(String email) {
        this.email = Optional.ofNullable(email);
    }
    public Optional<String> getEmail() {
return email;
    }
}

User user = new User(null);
String contact = user.getEmail().orElse("No email provided");
System.out.println(contact);

user.getEmail().ifPresent(e -> System.out.println("Sending to: " + 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. 8. Executor Service – Case Study: File Upload Service Scenario: You allow users to upload multiple files simultaneously and want to manage the processing efficiently.

```
LocalDate checkIn = LocalDate.of(2025, 8, 1);
LocalDate checkOut = LocalDate.of(2025, 8, 5);

// Duration
Period stayPeriod = Period.between(checkIn, checkOut);
System.out.println("Days: " + stayPeriod.getDays());

// Validate
if (checkOut.isBefore(checkIn)) {
    System.out.println("Invalid booking dates!");
}

// Recurring events
LocalDate recurring = LocalDate.now();
for (int i = 0; i < 3; i++) {
recurring = recurring.plusWeeks(1);
    System.out.println("Next booking: " + recurring); }
```

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.

```
ExecutorService executor = Executors.newFixedThreadPool(3);

Runnable uploadTask = () -> System.out.println("Uploading by " +
Thread.currentThread().getName());
 for (int i = 0; i < 5; i++) {
executor.submit(uploadTask);
}  executor.shutdown(); // Always shut
down!
```