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.

Program:

```
package Java8;
import java.util.Arrays;
import java.util.List;

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 LambdaExpression {
    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(
            new Employee("Sree", 26000),
            new Employee("Vani", 48000),
            new Employee("Dasari", 95000)
        );

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

        // Sort by salary
        employees.sort((e1, e2) -> Double.compare(e1.salary, e2.salary));
```

```
                System.out.println("Sorted by salary: " + employees);

        // Filter salary > 50000
        employees.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.

Program:

```
package Java8;

import java.util.Arrays;
import java.util.List;
import java.util.Map;
import 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 StreamAPI {

    public static void main(String[] args) {
        List<Order> orders = Arrays.asList(
            new Order("Sree", "Electronics", 12000),
            new Order("Vani", "Books", 800),
            new Order("Dasari", "Cosmetics", 1000), new
            Order("Tej", "Clothing", 4000), new
            Order("Bob", "Electronics", 15000)
        );

        // 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(Collectors.groupingBy(o -> o.customer,
Collectors.counting()));
        System.out.println("Order count: " + orderCount);

        // Group by category
        Map<String, List<Order>> grouped = orders.stream()
            .collect(Collectors.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`.

Program:

```
package Java8;


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

public class LoggerApp {
        public static void main(String[] args) {
      Predicate<String> errorFilter = msg -> msg.contains("Error");
      Consumer<String> logAction = msg -> System.out.println("LOG: "
+ msg);

      log("System started", errorFilter, logAction);
      log("Error: Unable to connect", errorFilter, logAction);
   }

   public static void log(String message, Predicate<String> filter,
Consumer<String> action) {
      if (filter.test(message)) {
         action.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.

Program:

```
package Java8;

public interface payment {
        void pay(double amount);

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

```
        }
    package Java8;

    public class Paypal implements payment {

        @Override
        public void pay(double amount) {
            System.out.println("Paid ₹" + amount + " using PayPal");
            logTransaction(amount);
        }

    }
    package Java8;

    public class UPI implements payment {

        @Override
        public void pay(double amount) {
            System.out.println("Paid ₹" + amount + " using UPI");
        logTransaction(amount);


        }

    }
package Java8;

public class PaymentApp {

      public static void main(String[] args) { payment
            paypal = new Paypal();
     paypal.pay(1500);

     payment upi = new UPI(); upi.pay(750);

      }

}
```

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.

Program:

```
Package Java8;
    public interface Notifier {
        void notify(String message);

    }
    package Java8;

    public class NotificationService {
        public void sendEmail(String message) {
            System.out.println("Sending Email: " + message);
        }

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

        public void sendPush(String message) {
            System.out.println("Sending Push Notification: " + message);
        }
}
    package Java8;

    public class NotificationApp {
        public static void main(String[] args) {
      NotificationService service = new NotificationService();

      // Method references to instance methods Notifier
      emailNotifier = service::sendEmail; Notifier smsNotifier
      = service::sendSMS; Notifier pushNotifier =
      service::sendPush;

      // Using the method references emailNotifier.notify("Welcome to our
      service!"); smsNotifier.notify("Your OTP is 123456");
      pushNotifier.notify("You have a new 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`.

Program:

```
package Java8;

import java.util.Optional;

public class User {
        private String name;
   private Optional<String> email;

   public User(String name, String email) {
      this.name = name;
      this.email = Optional.ofNullable(email);
   }

   public void printProfile() {
      System.out.println("Name: " + name);

      // Print email if present, otherwise show "Not provided" email.ifPresentOrElse(
         e -> System.out.println("Email: " + e),
         () -> System.out.println("Email not provided")
      );
   }

   public Optional<String> getEmail() {
      return email;
   }
}
package Java8;

public class UserProfile {

        public static void main(String[] args) {
                User user1 = new User("Tej", "Tej@gmail.com"); User user2 =
      new User("Reena", null);  // no email provided
```

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

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.

Program:

```
package Java8;

import java.time.LocalDate;
import java.time.Period;

public class BookSystem {

    public static void main(String[] args) {
            LocalDate checkIn = LocalDate.of(2025, 7, 25);
        LocalDate checkOut = LocalDate.of(2025, 7, 30);

        Period stay = Period.between(checkIn, checkOut);
        System.out.println("Stay Duration: " + stay.getDays() + " days");


        if (checkOut.isBefore(checkIn)) {
            System.out.println("Invalid check-out date");
        }


        LocalDate recurring = LocalDate.now().plusWeeks(1); System.out.println("Next
        maintenance: " + 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.

Program:

```
package Java8;

public class FileUploader implements Runnable {
        private String fileName;

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

        @Override
        public void run() {
                System.out.println("Uploading " + fileName + " on thread: "
+ Thread.currentThread().getName());

                try {
                    Thread.sleep(1000);  // 1 second per file (simulated)
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Upload complete: " + fileName);
                  }
}
package Java8;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```

```
public class FileUploadApp {

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

        // Simulate uploading multiple files executor.submit(new
        FileUploader("photo1.jpg")); executor.submit(new
        FileUploader("doc2.pdf")); executor.submit(new
        FileUploader("video3.mp4")); executor.submit(new
        FileUploader("notes4.txt")); executor.submit(new
        FileUploader("image5.png"));

        // Shut down the executor (no more new tasks)
        executor.shutdown();


          }

}
```