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

```
package CaseStudy;

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 EmployeeLambdaExample {
        public static void main(String[] args) {
      List<Employee> employees = Arrays.asList(
         new Employee("Alice", 50000),
         new Employee("Bob", 70000),
         new Employee("Charlie", 45000)
      );

      // Sort by salary using lambda
      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(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.

```
package CaseStudy;


import java.util.Arrays;
import java.util.List;
import java.util.Map;
import static java.util.stream.Collectors.*;
class Order {
  String customer;
  String category;
  double amount;


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


  public String toString() {
    return customer + " - " + category + " - " + amount;
  }
}
public class OrderStreamExample {
        public static void main(String[] args) {
    List<Order> orders = Arrays.asList(
      new Order("Alice", "Electronics", 2500),
        new Order("Bob", "Clothing", 1500),
        new Order("Alice", "Electronics", 3000)
    );

    // Filter orders above 2000
    orders.stream()
          .filter(o -> o.amount > 2000)
          .forEach(System.out::println);

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

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

```
package CaseStudy;
@FunctionalInterface
interface LogFilter {
  boolean filter(String message);
}
public class LoggerFunctionalInterfaceExample {
        public static void main(String[] args) {
      LogFilter errorFilter = msg -> msg.startsWith("ERROR");

      logMessage("INFO - All good", errorFilter);
      logMessage("ERROR - Something went wrong", errorFilter);
  }

  public static void logMessage(String message, LogFilter filter) {
      if (filter.filter(message)) {
         System.out.println("Logging: " + 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 CaseStudy;
interface Payment {
    void pay(double amount);


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


class PayPal implements Payment {
    public void pay(double amount) {
        System.out.println("Paid ₹" + amount + " via PayPal");
        logTransaction(amount);
    }
}
public class PaymentGatewayExample {
            public static void main(String[] args) {
        Payment payment = new PayPal();
        payment.pay(1000);
    }
}
```

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

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

    public static void sendSMS(String msg) {
        System.out.println("SMS sent: " + msg);
    }
}
public class NotificationMethodRefExample {
            public static void main(String[] args) {
        NotificationService service = new NotificationService();

        Runnable emailNotifier = () -> service.sendEmail("Hello!");
        Runnable smsNotifier = () -> NotificationService.sendSMS("Hi!");

        emailNotifier.run();
        smsNotifier.run();
    }
}
```

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.

```
package CaseStudy;

import java.util.Optional;

class User {
    Optional<String> email;

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

    public void printEmail() {
        email.ifPresentOrElse(
            e -> System.out.println("Email: " + e),
            () -> System.out.println("Email not provided")
        );
    }
}
public class UserOptionalExample {
            public static void main(String[] args) {
        User u1 = new User("abc@example.com");
        User u2 = new User(null);

        u1.printEmail();
        u2.printEmail();
    }
}
```

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.

```
package CaseStudy;

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

public class BookingDateExample {
            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 (checkIn.isAfter(checkOut)) {
            System.out.println("Invalid check-in/check-out dates");
        } else {
            System.out.println("Valid booking");
        }
    }
}
```

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

```
package CaseStudy;


import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;


public class FileUploadExecutorExample {
            public static void main(String[] args) {
        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();
    }
}
```