COMPUTING | BUSINESS | ENGINEERING

SE3040 – Application Framework

Worksheet 01

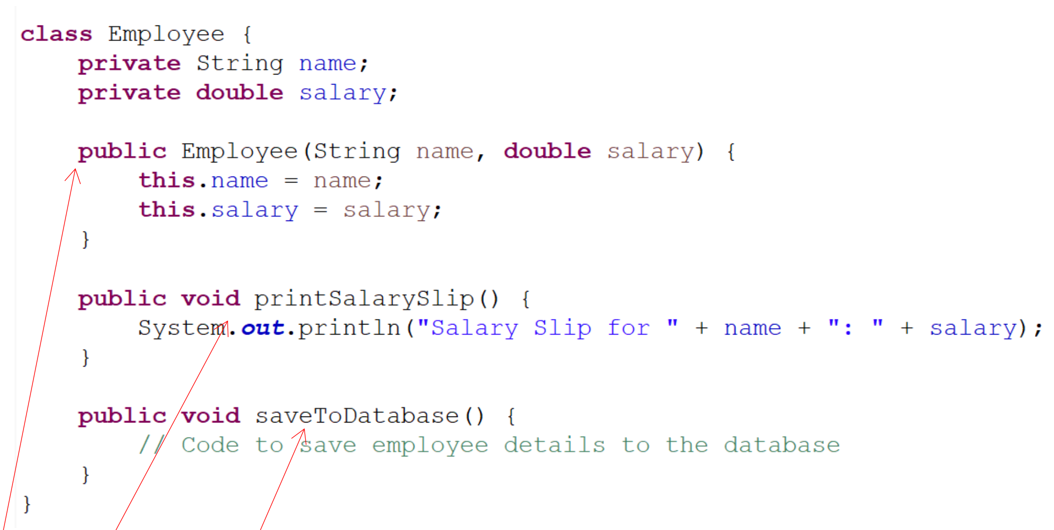
01. What are SOLID principles?

02. Understanding Single Responsibility Principle

a. What do you understand by Single Responsibility Principle?

A class should have one and only one reason to change, meaning that a class should have only one job.

b. Consider the following Java class that manages both employee data and report generation. What violation of the SOLID principles do you see in the above class?

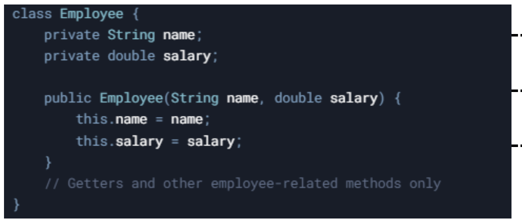


```
class Employee {
    private String name;
    private double salary;

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

    public void printSalarySlip() {
        System.out.println("Salary Slip for " + name + ": " + salary);
    }

    public void saveToDatabase() {
        // Code to save employee details to the database
    }
}
```

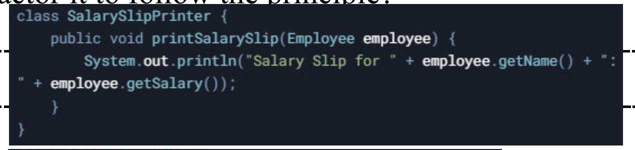
this class violate single-responsible-principle. this class has multiple responsibilities :
Employee data management (storing name and salary)
Report generation (printSalarySlip() method)
Database persistence (saveToDatabase() method)

The class should be refactored to separate these concerns

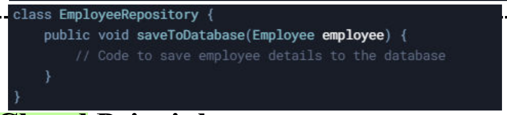
c. How would you refactor it to follow the principle?



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

    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }
    // Getters and other employee-related methods only
}
```



```
class SalarySlipPrinter {
    public void printSalarySlip(Employee employee) {
        System.out.println("Salary Slip for " + employee.getName() + ":
" + employee.getSalary());
    }
}
```



```
class EmployeeRepository {
    public void saveToDatabase(Employee employee) {
        // Code to save employee details to the database
    }
}
```

03. Understanding Open/Closed Principle

a. What do you understand by Open/Closed Principle?

Objects or entities should be open for extension, but closed for modification

b. Assume you have the following code snippet for a payment system. What violation of the Open/Closed Principle exists here?

```
class PaymentProcessor {
    public void processPayment(String type, double amount) {
        if (type.equals("CreditCard")) {
            // Process credit card payment
        } else if (type.equals("PayPal")) {
            // Process PayPal payment
        } else {
            System.out.println("Invalid payment method");
        }
    }
}
```

The paymentProcessor class violate open/closed principle because:

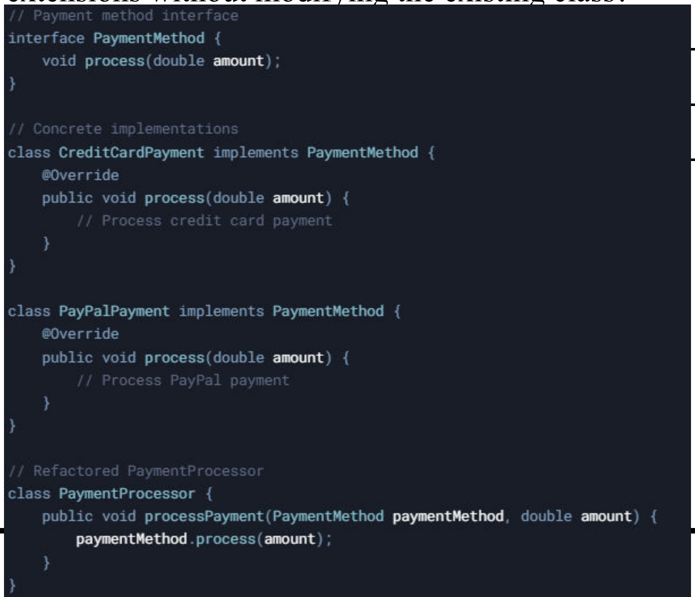
Requires modification of the PaymentProcessor class whenever a new payment method is added

this clsss use conditional logic (if-else) to handle different payment types

Is not closed for modification when requirements change

c. How can you modify the code to follow Open/Closed Principle, allowing future extensions without modifying the existing class?

"abstraction"
use for class extension



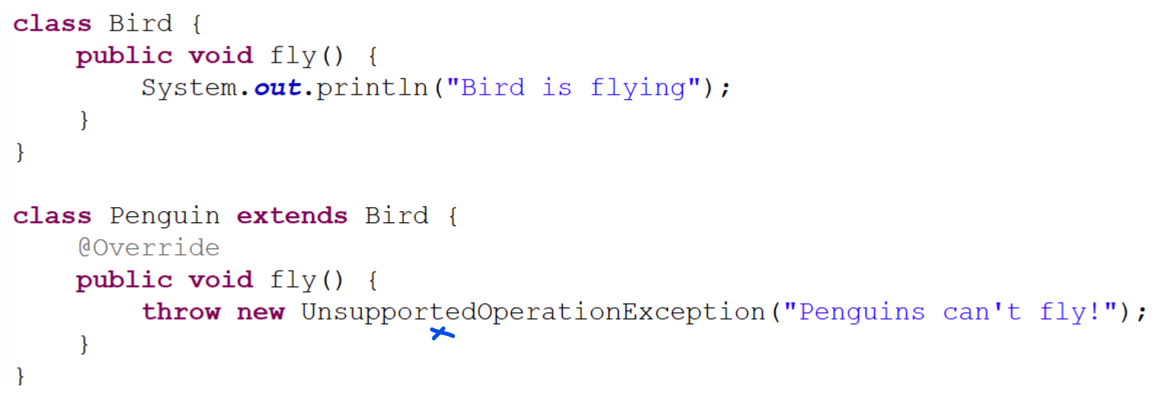
```
// Payment method interface
interface PaymentMethod {
    void process(double amount);
}

// Concrete implementations
class CreditCardPayment implements PaymentMethod {
    @Override
    public void process(double amount) {
        // Process credit card payment
    }
}

class PayPalPayment implements PaymentMethod {
    @Override
    public void process(double amount) {
        // Process PayPal payment
    }
}

// Refactored PaymentProcessor
class PaymentProcessor {
    public void processPayment(PaymentMethod paymentMethod, double amount) {
        paymentMethod.process(amount);
    }
}
```

04. Understanding Liskov Substitution Principle

a. What do you understand by Liskov Substitution Principle?

Every subclass/derived class should be able to substitute their parent/base class

b. Consider the following class hierarchy. Does this design comply with the Liskov Substitution Principle? State your reason?

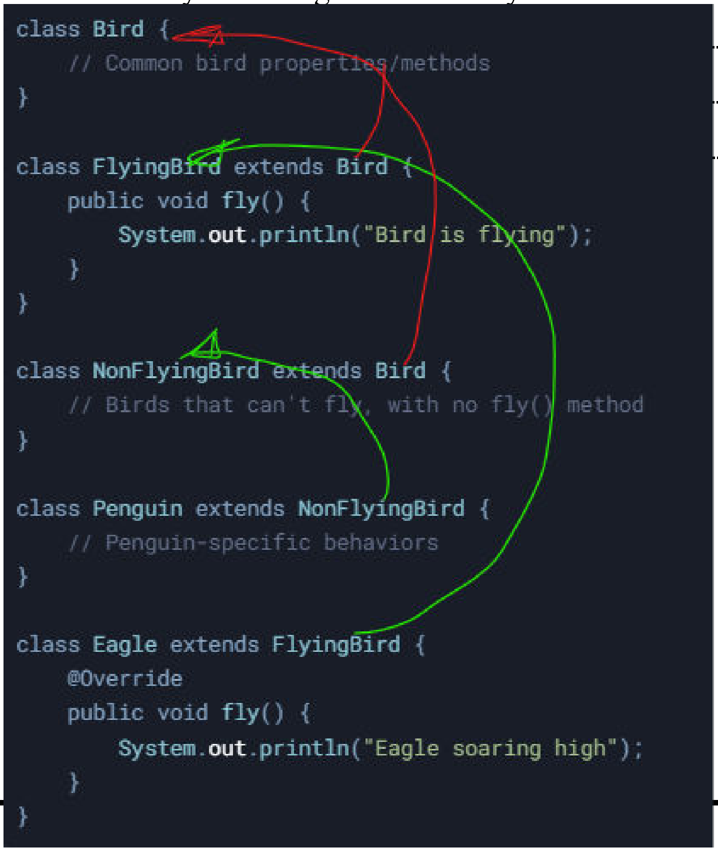


```java
class Bird {
    public void fly() {
        System.out.println("Bird is flying");
    }
}

class Penguin extends Bird {
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Penguins can't fly!");
    }
}
```

Penguin is a subclass of Bird but cannot fulfill the base class's contract. fly() method throws an exception instead of providing flying behavior

Why This Violates LSP,
Objects of a superclass should be replaceable with objects of its subclasses without breaking the application

c. How would you redesign the hierarchy to follow Liskov Substitution Principle?



```java
class Bird {
    // Common bird properties/methods
}

class FlyingBird extends Bird {
    public void fly() {
        System.out.println("Bird is flying");
    }
}

class NonFlyingBird extends Bird {
    // Birds that can't fly, with no fly() method
}

class Penguin extends NonFlyingBird {
    // Penguin-specific behaviors
}

class Eagle extends FlyingBird {
    @Override
    public void fly() {
        System.out.println("Eagle soaring high");
    }
}
```

05. Understanding the Interface segregation

a. What do you understand by Interface segregation?

Clients should not be forced to implement methods they do not use

----

----

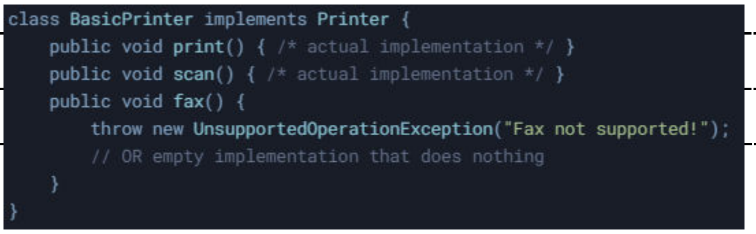
b. You are designing an interface for different types of printers. However, some printers only support printing and scanning but not faxing. How does this design **violate** the Interface Segregation Principle?

```
interface Printer {
    void print();
    void scan();
    void fax();
}
```

Printer interface is the fat interface (combining printing, scanning, and faxing)

Forces all printer implementations to support all three functions

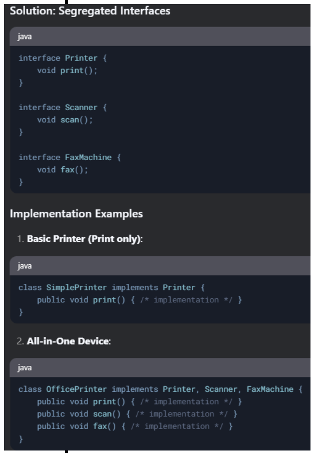
Many printers can't fax.



```
class BasicPrinter implements Printer {
    public void print() { /* actual implementation */ }
    public void scan() { /* actual implementation */ }
    public void fax() {
        throw new UnsupportedOperationException("Fax not supported!");
        // OR empty implementation that does nothing
    }
}
```

Runtime exceptions instead of compile-time safety

Suggest a better approach to redesign the interface to adhere to Interface segregation?



Solution: Segregated Interfaces

```java
interface Printer {
    void print();
}

interface Scanner {
    void scan();
}

interface FaxMachine {
    void fax();
}
```

Implementation Examples

1. **Basic Printer (Print only):**

```java
class SimplePrinter implements Printer {
    public void print() { /* implementation */ }
}
```

2. **All-in-One Device:**

```java
class OfficePrinter implements Printer, Scanner, FaxMachine {
    public void print() { /* implementation */ }
    public void scan() { /* implementation */ }
    public void fax() { /* implementation */ }
}
```

----

06. Understanding the Dependency inversion

a. What do you understand by Dependency inversion?

Higher level modules should not depend on lower level modules, but they should depend on abstractions

----

----

b. Examine the following dependency structure in a notification system. Why does this implementation **violate** the Dependency Inversion Principle?

High-level module (Notification) directly depends on low-level module (EmailService)

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

class Notification {
    private EmailService emailService;

    public Notification() {
        this.emailService = new EmailService();
    }

    public void sendNotification(String message) {
        emailService.sendEmail(message);
    }
}
```

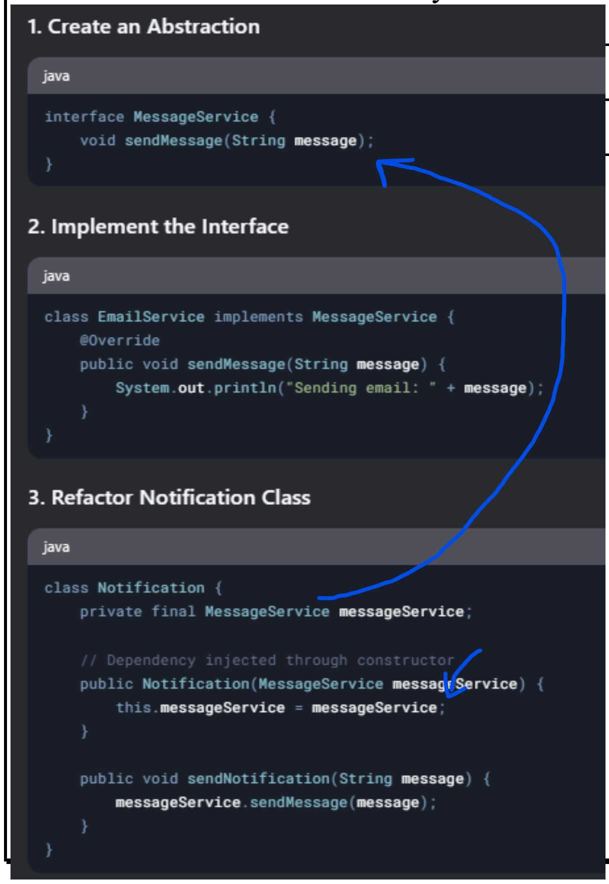
Notification directly instantiates EmailService

Makes it impossible to change notification mechanism without modifying Notification class

----

----

c. How would you refactor it using Dependency inversion?



1. Create an Abstraction

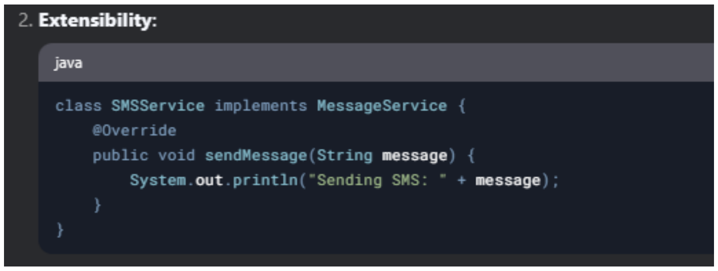
java
```java
interface MessageService {
    void sendMessage(String message);
}
```

2. Implement the Interface

java
```java
class EmailService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println("Sending email: " + message);
    }
}
```

3. Refactor Notification Class

java
```java
class Notification {
    private final MessageService messageService;

    // Dependency injected through constructor
    public Notification(MessageService messageService) {
        this.messageService = messageService;
    }

    public void sendNotification(String message) {
        messageService.sendMessage(message);
    }
}
```



2. **Extensibility:**

java
```java
class SMSService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println("Sending SMS: " + message);
    }
}
```