SOLID PRINCIPLES

1. Single Responsibility Principle (SRP):

In my program, all classes and interfaces follow this principle.

- **Payment** interface focuses solely on payment-related behaviors.
- **CardPayment** and **UpiPayment** classes implement payment logic specific to their payment methods.
- **PaymentProcessor** primarily handles payment processing logic.
- **OrderManager** and **Orders** handle order-related responsibilities.

```
public class PaymentProcessor {
    private OrderManager orderManager;

    public PaymentProcessor(OrderManager orderManager) {
        this.orderManager = orderManager;
    }

    public double calculateTax(double totalAmount) {
        return totalAmount * 0.1;  // Assuming 10% product tax
    }

    public double calculateShipping(double totalAmount) {
        return 4.0;  // Assuming a flat $5.0 shipping charge
    }
    public void processPayment(Payment payment) {
        //Print payment details
        System.out.println("Processing payment with ID: " + payment.getTotalAmount());
        // Add additional PAYMENT processing specific to the payment type.
        payment.processPayment();
        // Add to order after payment
        //adding id, total,payment type(currently using dummy value)
        Orders order = new Orders(0, payment.getTotalAmount(), null);
        orderManager.addOrder(order);
    }
}
```

```
public interface OrderManager {
void addOrder(Orders order);
void displayAllOrders();
}
```

2. Open/Closed Principle (OCP):

- The **PaymentProcessor** class is open for extension (new payment methods can be added without modifying existing code) and closed for modification.
- The introduction of the **OrderManager** interface allows for extending order-related functionalities without modifying existing code.

```java
public class PaymentProcessor {
    private OrderManager orderManager;

    public PaymentProcessor(OrderManager orderManager) {
        this.orderManager = orderManager;
    }

    public double calculateTax(double totalAmount) {
        return totalAmount * 0.1;  // Assuming 10% product tax
    }

    public double calculateShipping(double totalAmount) {
        return 4.0;  // Assuming a flat $5.0 shipping charge
    }
    public void processPayment(Payment payment) {
        //Print payment details
        System.out.println("Processing payment with ID: " + payment.getTotalAmount());
        // Add additional PAYMENT processing specific to the payment type.
        payment.processPayment();
        // Add to order after payment
        //adding id, total,payment type(currently using dummy value)
        Orders order = new Orders(0, payment.getTotalAmount(), null);
        orderManager.addOrder(order);
    }
}
```

3. Liskov Substitution Principle (LSP):

CardPayment and **UpiPayment** can be substituted for the **Payment** interface without affecting the correctness of the program.

CardPayment and **UpiPayment** can be substituted for the **Payment** interface without affecting the correctness of the program.

Card Payment class

```java
public class CardPayment implements Payment {
@Override
public void processPayment() {
// Card payment logic
System.out.println("Processing card payment: $" + getTotalAmount() + " with card number: " + cardNumber);
}
}
```

Upipayment class

```java
public class UpiPayment implements Payment {
@Override
public void processPayment() {
// Card payment logic
System.out.println("Processing UPI payment: $" + getTotalAmount() + " with upi id: " + upiId);
```

```
}
}
```

4. Interface Segregation Principle (ISP):

Interfaces (**Payment**, **OrderManager**) are designed to be specific and focused on a particular set of related behaviors.

```
public interface Payment {
void processPayment();
}

public interface OrderManager {
void addOrder(Orders order);
void displayAllOrders();
}
```

5. Dependency Inversion Principle (DIP):

High-level modules (e.g., **PaymentProcessor**) depend on abstractions (**Payment**, **OrderManager**), not on concrete implementations. This promotes flexibility and ease of extension.

PaymentProcessor (Depends on Abstractions):

```
public class PaymentProcessor {
private OrderManager orderManager;

public PaymentProcessor(OrderManager orderManager) {
this.orderManager = orderManager;
}
public double calculateTax(double totalAmount) {
return totalAmount * 0.1; // Assuming 10% product tax
}
public double calculateShipping(double totalAmount) {
return 4.0; // Assuming a flat $5.0 shipping charge
}
public void processPayment(Payment payment) {
//Print payment details
System.out.println("Processing payment with ID: " + payment.getTotalAmount());
// Add additional PAYMENT processing specific to the payment type.
payment.processPayment();
```

```
// Add to order after payment
//adding id, total,payment type(currently using dummy value)
Orders order = new Orders(0, payment.getTotalAmount(), null);
orderManager.addOrder(order);
}
}
```