SOLID PRINCIPLES

1. What is the dependency inversion principle? Explain how it contributes to the more testable code.

```java
package w6w6;

    //The dependancy inversion principle is a software development
    //concept that helps make code more flexible and maintainable.
    //it states that :
    //high level modules must not depend on low level modules.
    //abstractions should not depend on details; details
    //should depend on abstractions.

class EmailService1 {
    public void sendEmail(String message) {
        System.out.println("Sending email: "+ message);
    }
}
//Violating DIP
class Notification1{
    private EmailService1 emailService;

    public Notification1(EmailService EmailService){
        this.emailService= new EmailService1();
    }
    public void notify(String message) {

        emailService.sendEmail(message);
    }
}
```

SOLID PRINCIPLES

```java
interface NotificationService{
    void sendNotification(String message);
}
class EmailService implements NotificationService{
    @Override
    public void sendNotification(String message) {
        System.out.println("Sending SMS: "+message);
    }
}
class SMSService implements NotificationService{
    @Override
    public void sendNotification(String message) {
        System.out.println("Sending SMS: "+message);
    }
}
class Notification{
    private NotificationService notificationService;

    public Notification(NotificationService notificationService){
        this.notificationService= notificationService;
    }
    public void notify(String message) {
        notificationService.sendNotification(message);
    }
}
```

2. Describe the scenario where applying the Open-Closed Principle leads to improved code quality.

```java
class ShapeDrawer1{ //violating Open closed principle
    public void drawShape(String shapeType) {
        if (shapeType.equals("Circle")) {
            System.out.println("Drawing a Lasta (circle)");
        }else if (shapeType.equals("Rectangle")) {
            System.out.println("Drawing a Salina (rectangle)");
        }
    }
}
```

SOLID PRINCIPLES

```
interface Shape{
    void draw();
}
class Circle implements Shape{
    @Override
    public void draw() {
        System.out.println("Drawing a Lasta (circle)");
    }
}
class Rectangle implements Shape{
    @Override
    public void draw() {
        System.out.println("Drawing a Salina(rectangle)");
    }
}
class ShapeDrawer{
    public void drawShape(Shape shape) {
        shape.draw();
    }
}
public class w6q2{
    public static void main(String[] args) {
        ShapeDrawer shapedrawer = new ShapeDrawer();

        Shape circle = new Circle();
        Shape rectangle = new Rectangle();

        shapedrawer.drawShape(circle);
        shapedrawer.drawShape(rectangle);
    }
}
```

3. Explain the scenario where the Interface Segregation Principle was beneficial.

SOLID PRINCIPLES

```java
//Interface segregation principle states that no client
//should be forced to depend on methods it does not use.

interface BorrowOperations { // separate interface for borrowing
    void borrowBook();
    void returnBook();
}
interface ManageBooks { //separate interface for managing
    void addBook();
    void removeBook();
}
class Student implements BorrowOperations {
    @Override
    public void borrowBook() {
        System.out.println("Borrowing a book...");
    }

    @Override
    public void returnBook() {
        System.out.println("Returning a book...");
    }
}

class Librarian implements ManageBooks, BorrowOperations {
    @Override
    public void addBook() {
        System.out.println("Adding a book...");
    }
    @Override
    public void removeBook() {
        System.out.println("Removing a book...");
    }
    @Override
    public void borrowBook() {
        System.out.println("Borrowing a book...");
    }
    @Override
    public void returnBook() {
        System.out.println("Returning a book...");
    }
}
```

4. Examine the following code.

```
public class Report {
  public void generateReport() {
    // generate report logic
  }

  public void exportToPDF() {
    // export report to PDF logic
  }

  public void exportToExcel() {
    // export report to Excel logic
  }
}
```

Which principle is violated in the code among Single Responsibility, Open Closed, Interface Segregation, and Dependency Inversion Principles? Explain in detail.

SOLID PRINCIPLES

```java
package w6w6;

//This code violates the Single Responsibility Principle (SRP)
//because the Report class handles both report generation and
//exporting logic. By separating these concerns into different
//classes, you can create a more maintainable, flexible, and
//modular system.

class Report1 {
    public void generateReport() {
       System.out.println("Generating report..");
       // generate report logic
     }
    public void exportToPDF() {
        System.out.println("Exporting to pdf..");
        // export report to PDF logic
     }

     public void exportToExcel() {
         System.out.println("Exporting to excel..");
        // export report to Excel logic
     }
}
```

```java
//not violating SRP:

//Class responsible for generating reports
class Report {
    public void generateReport() {
   // generate report logic
     }
}

//Interface for exporting reports
interface ReportExporter {
    void export(Report report);
}

//Class responsible for exporting to PDF
class PDFExporter implements ReportExporter {
    @Override
        public void export(Report report) {
        // export report to PDF logic
    }
}

//Class responsible for exporting to Excel
class ExcelExporter implements ReportExporter {
    @Override
    public void export(Report report) {
        // export report to Excel logic
    }
}
```

SOLID PRINCIPLES

5. Can you provide an example of how to design an online payment processing system while adhering to the SOLID principles? Please explain how each principle can be applied in the context of this system and illustrate with code or a conceptual overview. Let's assume we have payment types like CreditCardPayment, PayPalPayment, Esewa, and Khalti. Each of these payments should have a method of transferring the amount.

```java
package w6w6;

// Adhering the SOLID principles can help create modular, flexible
//and maintainable system.

//PaymentProcessor Interface: Defines the abstraction for all payment types
interface PaymentProcessor {
    void processPayment(double amount);
}

//CreditCardPayment: Implements the PaymentProcessor for Credit Card payments
class CreditCardPayment implements PaymentProcessor {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing credit card payment of $" + amount);
    }
}

//PayPalPayment: Implements the PaymentProcessor for PayPal payments
class PayPalPayment implements PaymentProcessor {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing PayPal payment of $" + amount);
    }
}

//EsewaPayment: Implements the PaymentProcessor for Esewa payments
class EsewaPayment implements PaymentProcessor {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing Esewa payment of $" + amount);
    }
}

//KhaltiPayment: Implements the PaymentProcessor for Khalti payments
class KhaltiPayment implements PaymentProcessor {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing Khalti payment of $" + amount);
    }
}
```

SOLID PRINCIPLES

```java
//PaymentLogger: A separate class for logging (adhering to Single Responsibility Principle)
class PaymentLogger {
    public void log(String message) {
        System.out.println("Log: " + message);
    }
}

//PaymentNotifier: A separate class for sending notifications (adhering to SRP)
class PaymentNotifier {
    public void sendNotification(String message) {
        System.out.println("Notification: " + message);
    }
}

//PaymentService: High-level class responsible for processing payments
class PaymentService {
    private PaymentProcessor paymentProcessor;
    private PaymentLogger paymentLogger;
    private PaymentNotifier paymentNotifier;

     // Constructor Dependency Injection
     public PaymentService(PaymentProcessor paymentProcessor) {
         this.paymentProcessor = paymentProcessor;
         this.paymentLogger = new PaymentLogger(); // Separate responsibility for logging
         this.paymentNotifier = new PaymentNotifier(); // Separate responsibility for notifications
     }

     // Process the payment and handle logging and notification
     public void processPayment(double amount) {
         paymentProcessor.processPayment(amount);
         paymentLogger.log("Payment of $" + amount + " processed.");
         paymentNotifier.sendNotification("Payment of $" + amount + " successful.");
     }
}
```

```java
//Main class to demonstrate the system (this can be public since it has the main method)
public class w6q5 {
    public static void main(String[] args) {
        // Creating payment processor objects for different payment methods
        PaymentProcessor creditCardPayment = new CreditCardPayment();
        PaymentProcessor payPalPayment = new PayPalPayment();
        PaymentProcessor esewaPayment = new EsewaPayment();
        PaymentProcessor khaltiPayment = new KhaltiPayment();

        // Processing payments using different payment methods
        PaymentService creditCardService = new PaymentService(creditCardPayment);
        creditCardService.processPayment(100.0); // Processing credit card payment

         PaymentService payPalService = new PaymentService(payPalPayment);
         payPalService.processPayment(200.0); // Processing PayPal payment

         PaymentService esewaService = new PaymentService(esewaPayment);
         esewaService.processPayment(300.0); // Processing Esewa payment

         PaymentService khaltiService = new PaymentService(khaltiPayment);
         khaltiService.processPayment(400.0); // Processing Khalti payment
     }
}
```

6. Examine the following code.

```
public class Shape {
  public void drawCircle() {
   // drawing circle logic
  }
```

```
    public void drawSquare() {
     // drawing square logic
    }
}
```

You want to add more shapes (e.g., triangles, rectangles) without modifying the existing Shape class. Which design change would adhere to the Open-Closed Principle?

```java
package w6w6;

class Shape {
    public void drawCircle() {
        System.out.println("Drawing circle");
    }

    public void drawSquare() {
        System.out.println("Drawing square");
    }
}
//new interface for new shapes without changing shape class
interface newShape{
    void draw();
}
class triangle implements newShape{
    public void draw() {
        System.out.println("Drawing triangle");
    }
}
class rectangle implements newShape{
    public void draw() {
        System.out.println("Drawing rectangle");
    }
}
public class w6q6{
    public static void main(String[] args) {
        Shape shape = new Shape();
        triangle t = new triangle();
        rectangle r = new rectangle();
        shape.drawCircle();
        shape.drawSquare();
        t.draw();
        r.draw();
    }
}
```

```
Drawing circle
Drawing square
Drawing triangle
Drawing rectangle
```

SOLID PRINCIPLES

7. Examine the following code.

```
public class Duck {
  public void swim() {
    System.out.println("Swimming");
  }
  public void quack() {
    System.out.println("Quacking");
  }
}

public class WoodenDuck extends Duck {
  @Override
  public void quack() {
    throw new UnsupportedOperationException("Wooden ducks don't quack");
  }
}
```

Which principle is violated in the above code among Open Closed, Single Responsibility, Liskov, and Interface Segregation Principle? Explain in detail. Also, update the above code base to resolve the issue.

```
class Duck1 {
    public void swim() {
        System.out.println("Swimming");
    }
    public void quack() {
        System.out.println("Quacking");
    }
}

class WoodenDuck1 extends Duck1 {
    @Override
    public void quack() {
        throw new UnsupportedOperationException("Wooden ducks don't quack");
    }
}
```

SOLID PRINCIPLES

```java
//correct code:
//Interface for ducks that can swim
interface Swimmable {
    void swim();
}

//Interface for ducks that can quack
interface Quackable {
    void quack();
}
//Duck class that can swim and quack
class Duck implements Swimmable, Quackable {
    @Override
    public void swim() {
        System.out.println("Swimming");
    }

    @Override
    public void quack() {
        System.out.println("Quacking");
    }
}

//WoodenDuck class that can swim but does not quack
class WoodenDuck implements Swimmable {
    @Override
    public void swim() {
        System.out.println("Swimming");
    }
}
```

```java
//MallardDuck class that can swim and quack
class MallardDuck implements Swimmable, Quackable {
    @Override
    public void swim() {
        System.out.println("Swimming");
    }

    @Override
    public void quack() {
        System.out.println("Quacking");
    }
}
```

SOLID PRINCIPLES

8. Examine the following code.

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

public class PaypalPayment implements PaymentMethod {
  @Override
  public void processPayment() {
    System.out.println("Processing PayPal payment");
  }
}

public class OrderService {

  private PaymentMethod paymentMethod;

  public OrderService(PaymentMethod paymentMethod) {
    this.paymentMethod = paymentMethod;
  }

  public void makePayment() {
    paymentMethod.processPayment();
  }
}
```

Which solid principle is being used in above? Explain in detail.

SOLID PRINCIPLES

```java
package w6w6;

//the SOLID principle being used in this code is Dependency Inversion Principle
//which states that high level modules should not depend on low level modules.

interface PaymentMethod {
    void processPayment();
}

class PaypalPayment implements PaymentMethod {
    @Override
    public void processPayment() {
        System.out.println("Processing PayPal payment");
    }
}

class OrderService { //orderService method is not dependent on paypalMethod

    private PaymentMethod paymentMethod;

    public OrderService(PaymentMethod paymentMethod) {
        this.paymentMethod = paymentMethod;
    }

    public void makePayment() {
        paymentMethod.processPayment();
    }
}
//In this code the orderService method depends on PaymentMethod and not on concrete
//class like PaypalPayment method. this makes the code flexible and manageable.
```