1. **What we mean by coding against interface rather than class ? and if u get it so What we mean by code against abstraction not concreteness ?**

**the principles of "coding against interfaces, not classes" and "coding against abstraction, not concreteness" are equally applicable. Here's how they translate**

**Coding Against Interfaces, Not Classes**

**This means designing your code to rely on interfaces (interface) instead of specific class implementations. Interfaces define a contract, allowing multiple classes to implement the same behavior while keeping the consuming code flexible and decoupled.**

**Example:**

**// Interface defining the contract**

**public interface IPaymentProcessor**

**{**

**void ProcessPayment(decimal amount);**

**}**

**// Concrete implementation of the interface**

**public class PayPalProcessor : IPaymentProcessor**

**{**

**public void ProcessPayment(decimal amount)**

**{**

**Console.WriteLine($"Processing payment of {amount:C} through PayPal.");**

**}**

**}**

**public class StripeProcessor : IPaymentProcessor**

**{**

**public void ProcessPayment(decimal amount)**

**{**

**Console.WriteLine($"Processing payment of {amount:C} through Stripe.");**

**}**

**}**

**// Service depending on the abstraction (interface)**

**public class PaymentService**

**{**

**private readonly IPaymentProcessor \_paymentProcessor;**

**public PaymentService(IPaymentProcessor paymentProcessor)**

**{**

**\_paymentProcessor = paymentProcessor;**

**}**

**public void MakePayment(decimal amount)**

**{**

**\_paymentProcessor.ProcessPayment(amount);**

**}**

**}**

**// Usage**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**IPaymentProcessor processor = new PayPalProcessor();**

**PaymentService service = new PaymentService(processor);**

**service.MakePayment(100.00m);**

**processor = new StripeProcessor();**

**service = new PaymentService(processor);**

**service.MakePayment(150.00m);**

**}**

**}**

**Benefits :**

1. **Loose Coupling: PaymentService works with any implementation of IPaymentProcessor, not just one specific class.**
2. **Testability: You can pass a mock implementation of IPaymentProcessor during unit testing.**
3. **Extensibility: New payment processors can be added without modifying PaymentService.**

**Coding Against Abstraction, Not Concreteness in C#**

**This principle is broader and advocates for designing systems around abstractions (e.g., interfaces or abstract classes) rather than concrete implementations. It aligns with the Dependency Inversion Principle of SOLID design.**

**Example:**

**Imagine a drawing application where you render different shapes.**

**Bad (Concrete):**

**public class Circle**

**{**

**public void Draw()**

**{**

**Console.WriteLine("Drawing a circle.");**

**}**

**}**

**public class DrawingApp**

**{**

**private readonly Circle \_circle;**

**public DrawingApp()**

**{**

**\_circle = new Circle();**

**}**

**public void Render()**

**{**

**\_circle.Draw();**

**}**

**}**

**Here, DrawingApp directly depends on Circle. If you wanted to add a Square, you'd need to modify DrawingApp, violating the Open/Closed Principle.**

**Good (Abstract):**

**// Define an abstraction**

**public interface IShape**

**{**

**void Draw();**

**}**

**// Implement different shapes**

**public class Circle : IShape**

**{**

**public void Draw()**

**{**

**Console.WriteLine("Drawing a circle.");**

**}**

**}**

**public class Square : IShape**

**{**

**public void Draw()**

**{**

**Console.WriteLine("Drawing a square.");**

**}**

**}**

**// High-level module depends on the abstraction**

**public class DrawingApp**

**{**

**private readonly IShape \_shape;**

**public DrawingApp(IShape shape)**

**{**

**\_shape = shape;**

**}**

**public void Render()**

**{**

**\_shape.Draw();**

**}**

**}**

**// Usage**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**IShape circle = new Circle();**

**DrawingApp app = new DrawingApp(circle);**

**app.Render();**

**IShape square = new Square();**

**app = new DrawingApp(square);**

**app.Render();**

**}**

**}**

1. **Interface Segregation: Interfaces define the minimum contract; consumers only know what they need.**
2. **Dependency Injection: Dependencies (like IShape or IPaymentProcessor) are injected, making the system modular.**
3. **Flexibility: Adding a new shape like Triangle requires no changes to DrawingApp.**