Part 02

**2- 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?**

**Coding Against Interface Rather than Class**

**Coding against an interface** means designing your code to depend on abstractions (interfaces) rather than concrete implementations (classes). Here's why this is beneficial:

1. **Flexibility**: It allows you to change the implementation without changing the code that uses the interface.
2. **Decoupling**: It reduces dependencies between components, making the system more modular and easier to maintain.
3. **Testability**: It allows you to easily mock or stub implementations for unit testing.

**Example**: Instead of directly using a Car class, you might code against an IVehicle interface:

public interface IVehicle

{

void StartEngine();

void StopEngine();

}

public class Car : IVehicle

{

public void StartEngine() { /\* Implementation \*/ }

public void StopEngine() { /\* Implementation \*/ }

}

public class Bike : IVehicle

{

public void StartEngine() { /\* Implementation \*/ }

public void StopEngine() { /\* Implementation \*/ }

}

// Code against the interface

public void OperateVehicle(IVehicle vehicle)

{

vehicle.StartEngine();

vehicle.StopEngine();

}

**Code Against Abstraction, Not Concreteness**

**Abstraction** refers to the practice of exposing only the essential features of an object, hiding the implementation details. **Coding against abstraction** means you use interfaces or abstract classes rather than concrete classes. This makes your code more flexible and adaptable to change.

**Example**: Using the IVehicle interface in the previous example instead of specific classes (Car or Bike), your method can operate on any IVehicle implementation:

public void TestVehicle(IVehicle vehicle)

{

vehicle.StartEngine();

vehicle.StopEngine();

}

**3- What is abstraction as a guideline and how we can implement this through what we have studied?**

**Abstraction as a Guideline**

**Abstraction** is a fundamental principle in object-oriented design that allows you to manage complexity by breaking down the system into smaller, more manageable pieces. It involves:

1. **Hiding Implementation Details**: Showing only what is necessary.
2. **Exposing Functionality**: Through well-defined interfaces or abstract classes.

**Implementing Abstraction**

**From What We've Studied**:

1. **Interfaces**: Define common behaviors for different classes.
   * **Example**: IVehicle interface for Car and Bike.
2. **Abstract Classes**: Provide a base class with some common functionality, while leaving some methods to be implemented by derived classes.
   * **Example**: GeometricShape abstract class with derived classes Triangle and Rectangle.

**Example**:

// Interface abstraction

public interface ILogger

{

void Log();

}

// Abstract class abstraction

public abstract class Shape

{

public abstract double GetArea();

public void Display() => Console.WriteLine("Displaying shape");

}

By using abstraction, you create a flexible and scalable architecture where changes in implementation details don't impact the higher-level logic.