Exercise 1: Implementing the Singleton Pattern

Scenario:

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

```
public class Logger
{
    private static Logger instance;
    private static readonly object lockObj = new object();

    private Logger()
    {
        Console.WriteLine("Logger initialized.");
    }
```

```
    public static Logger GetInstance()
    {
        if (instance == null)
        {
            lock (lockObj)
            {
                if (instance == null)
                {
                    instance = new Logger();
                }
            }
        }
        return instance;
    }
```

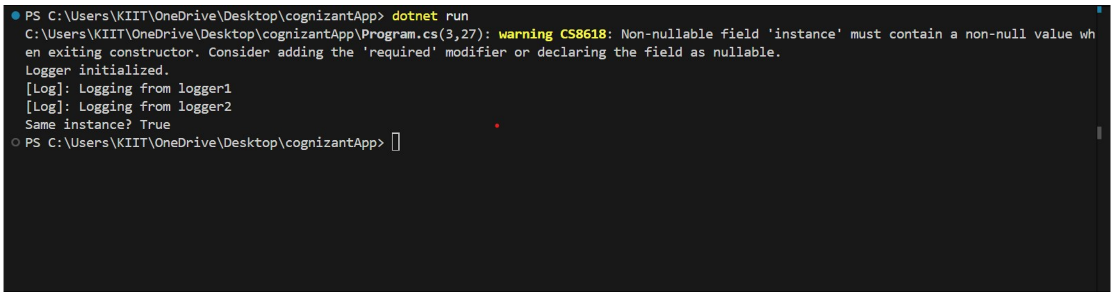
```
    public void Log(string message)
    {
        Console.WriteLine($"[Log]: {message}");
    }
}
```

```
public class Program
{
    public static void Main(string[] args)
    {
        Logger logger1 = Logger.GetInstance();
        Logger logger2 = Logger.GetInstance();
```

```
        logger1.Log("Logging from logger1");
        logger2.Log("Logging from logger2");
```

```
        Console.WriteLine($"Same instance? {ReferenceEquals(logger1, logger2)}");
    }
}
```

Output:



```
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp> dotnet run
C:\Users\KIIT\OneDrive\Desktop\cognizantApp\Program.cs(3,27): warning CS8618: Non-nullable field 'instance' must contain a non-null value wh
en exiting constructor. Consider adding the 'required' modifier or declaring the field as nullable.
Logger initialized.
[Log]: Logging from logger1
[Log]: Logging from logger2
Same instance? True
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp>
```

Exercise 2: Implementing the Factory Method Pattern

Scenario:

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

```
public interface IDocument
{
    void Open();
}

public class WordDocument : IDocument
{
    public void Open() => Console.WriteLine("Word document opened.");
}
```

```
public class PdfDocument : IDocument
{
    public void Open() => Console.WriteLine("PDF document opened.");
}
```

```
public class ExcelDocument : IDocument
{
    public void Open() => Console.WriteLine("Excel document opened.");
}
```

```
public abstract class DocumentFactory
{
    public abstract IDocument CreateDocument();
}
```

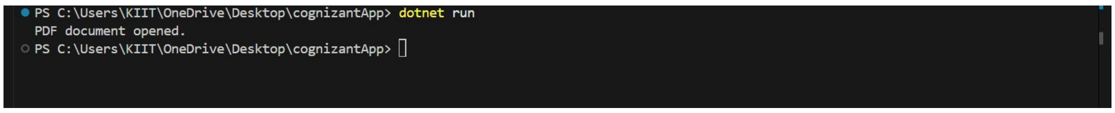
```
public class WordFactory : DocumentFactory
{
    public override IDocument CreateDocument() => new WordDocument();
}
```

```
public class PdfFactory : DocumentFactory
{
    public override IDocument CreateDocument() => new PdfDocument();
}
```

```
public class ExcelFactory : DocumentFactory
{
    public override IDocument CreateDocument() => new ExcelDocument();
}
```

```
public class Program
{
    public static void Main(string[] args)
    {
        DocumentFactory factory = new PdfFactory();
        IDocument doc = factory.CreateDocument();
        doc.Open();
    }
}
```

Output:



```
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp> dotnet run
PDF document opened.
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp>
```

Exercise 3: Implementing the Builder Pattern

Scenario:

You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

```
public class Computer
{
    public string CPU { get; }
    public string RAM { get; }
    public string Storage { get; }

    private Computer(Builder builder)
    {
        CPU = builder.CPU;
        RAM = builder.RAM;
        Storage = builder.Storage;
    }
```

```
    public class Builder
    {
        public string CPU;
        public string RAM;
        public string Storage;
```

```
        public Builder SetCPU(string cpu)
        {
            CPU = cpu;
            return this;
        }
```

```
        public Builder SetRAM(string ram)
        {
            RAM = ram;
            return this;
        }
```

```
        public Builder SetStorage(string storage)
        {
            Storage = storage;
            return this;
        }
```

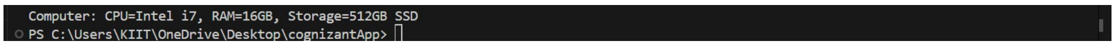
```
        public Computer Build()
        {
            return new Computer(this);
        }
    }
}
```

```
public class Program
{
    public static void Main(string[] args)
    {
        var computer = new Computer.Builder()
            .SetCPU("Intel i7")
            .SetRAM("16GB")
            .SetStorage("512GB SSD")
            .Build();
```

```
        Console.WriteLine($"Computer: CPU={computer.CPU}, RAM={computer.RAM}, Storage={computer.Storage}");
    }
```

```
}
```

Output:



```
Computer: CPU=Intel i7, RAM=16GB, Storage=512GB SSD
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp>
```

Exercise 4: Implementing the Adapter Pattern

Scenario:

You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

Steps:

```
public interface IPaymentProcessor
{
    void ProcessPayment();
}

public class StripeGateway
{
    public void MakeStripePayment() => Console.WriteLine("Payment made via Stripe.");
}
```

```
public class PayPalGateway
{
    public void PayWithPayPal() => Console.WriteLine("Payment made via PayPal.");
}
```

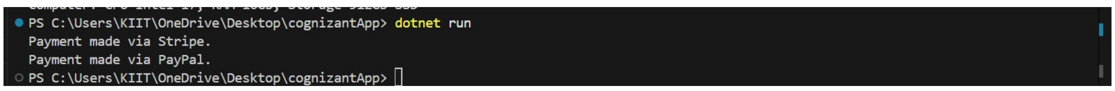
```
public class StripeAdapter : IPaymentProcessor
{
    private StripeGateway _stripe;
    public StripeAdapter(StripeGateway stripe) => _stripe = stripe;
    public void ProcessPayment() => _stripe.MakeStripePayment();
}
```

```
public class PayPalAdapter : IPaymentProcessor
{
    private PayPalGateway _paypal;
    public PayPalAdapter(PayPalGateway paypal) => _paypal = paypal;
    public void ProcessPayment() => _paypal.PayWithPayPal();
}
```

```
public class Program
{
    public static void Main(string[] args)
    {
        IPaymentProcessor stripe = new StripeAdapter(new StripeGateway());
        IPaymentProcessor paypal = new PayPalAdapter(new PayPalGateway());
```

```
        stripe.ProcessPayment();
        paypal.ProcessPayment();
    }
}
```

Output:



```
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp> dotnet run
Payment made via Stripe.
Payment made via PayPal.
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp>
```

Exercise 5: Implementing the Decorator Pattern

Scenario:

You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

```
public interface INotifier
{
    void Send(string message);
}

public class EmailNotifier : INotifier
{
    public void Send(string message)
    {
        Console.WriteLine($"Email: {message}");
    }
}
```

```
public abstract class NotifierDecorator : INotifier
{
    protected INotifier wrappee;
```

```
    public NotifierDecorator(INotifier notifier)
    {
        wrappee = notifier;
    }
```

```
    public virtual void Send(string message)
    {
        wrappee.Send(message);
    }
}
```

```
public class SMSNotifier : NotifierDecorator
{
    public SMSNotifier(INotifier notifier) : base(notifier) { }
```

```
    public override void Send(string message)
    {
        base.Send(message);
        Console.WriteLine($"SMS: {message}");
    }
}
```

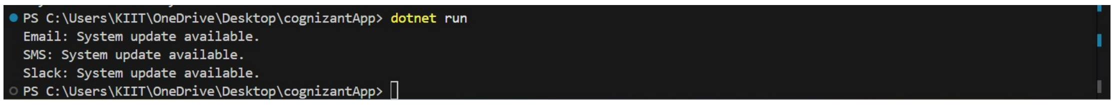
```
public class SlackNotifier : NotifierDecorator
{
    public SlackNotifier(INotifier notifier) : base(notifier) { }
```

```
    public override void Send(string message)
    {
        base.Send(message);
        Console.WriteLine($"Slack: {message}");
    }
}
```

```
public class Program
{
    public static void Main(string[] args)
    {
        INotifier notifier = new EmailNotifier();
        notifier = new SMSNotifier(notifier);
        notifier = new SlackNotifier(notifier);
```

```
        notifier.Send("System update available.");
    }
}
```

Output:



```
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp> dotnet run
Email: System update available.
SMS: System update available.
Slack: System update available.
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp>
```

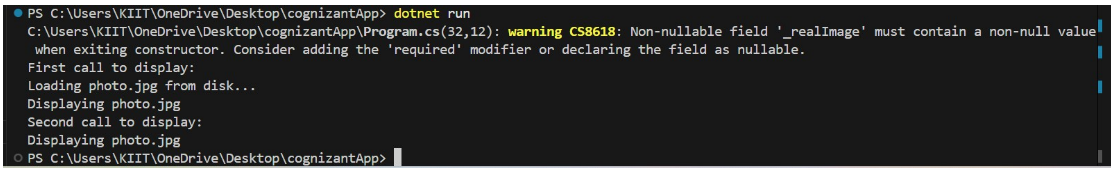
Exercise 6: Implementing the Proxy Pattern

Scenario:

You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

```
public interface IImage
{
    void Display();
}

public class RealImage : IImage
{
    private string _filename;

    public RealImage(string filename)
    {
        _filename = filename;
        LoadFromDisk();
    }

    private void LoadFromDisk()
    {
        Console.WriteLine($"Loading {_filename} from disk...");
    }

    public void Display()
    {
        Console.WriteLine($"Displaying {_filename}");
    }
}

public class ProxyImage : IImage
{
    private RealImage _realImage;
    private string _filename;

    public ProxyImage(string filename)
    {
        _filename = filename;
    }

    public void Display()
    {
        if (_realImage == null)
        {
            _realImage = new RealImage(_filename);
        }
        _realImage.Display();
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        IImage image = new ProxyImage("photo.jpg");

        Console.WriteLine("First call to display:");
        image.Display();
```

```
        Console.WriteLine("Second call to display:");
        image.Display();
    }
}
```



```
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp> dotnet run
C:\Users\KIIT\OneDrive\Desktop\cognizantApp\Program.cs(32,12): warning CS8618: Non-nullable field '_realImage' must contain a non-null value
 when exiting constructor. Consider adding the 'required' modifier or declaring the field as nullable.
First call to display:
Loading photo.jpg from disk...
Displaying photo.jpg
Second call to display:
Displaying photo.jpg
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp>
```

Exercise 7: Implementing the Observer Pattern

Scenario:

You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

```
using System;
using System.Collections.Generic;

public interface IObserver
{
    void Update(string stock, double price);
}
```

```
public interface IStock
{
    void RegisterObserver(IObserver observer);
    void RemoveObserver(IObserver observer);
    void NotifyObservers();
}
```

```
public class StockMarket : IStock
{
    private List<IObserver> observers = new List<IObserver>();
    private string stock;
    private double price;
```

```
    public void SetStockPrice(string stock, double price)
    {
        this.stock = stock;
        this.price = price;
        NotifyObservers();
    }
```

```
    public void RegisterObserver(IObserver observer)
    {
        observers.Add(observer);
    }
```

```
    public void RemoveObserver(IObserver observer)
    {
        observers.Remove(observer);
    }
```

```
    public void NotifyObservers()
    {
        foreach (var observer in observers)
        {
            observer.Update(stock, price);
        }
    }
}
```

```
public class MobileApp : IObserver
```

```
{
    public void Update(string stock, double price)
    {
        Console.WriteLine($"[Mobile] {stock} is now ${price}");
    }
}
```

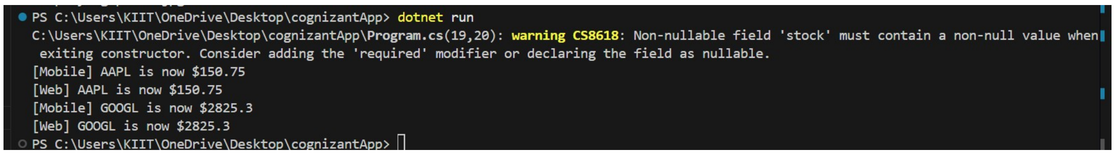
```
public class WebApp : IObserver
{
    public void Update(string stock, double price)
    {
        Console.WriteLine($"[Web] {stock} is now ${price}");
    }
}
```

```
public class Program
{
    public static void Main(string[] args)
    {
        StockMarket market = new StockMarket();
        IObserver mobile = new MobileApp();
        IObserver web = new WebApp();
```

```
        market.RegisterObserver(mobile);
        market.RegisterObserver(web);
```

```
        market.SetStockPrice("AAPL", 150.75);
        market.SetStockPrice("GOOGL", 2825.30);
    }
}
```

Output:



```
● PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp> dotnet run
  C:\Users\KIIT\OneDrive\Desktop\cognizantApp\Program.cs(19,20): warning CS8618: Non-nullable field 'stock' must contain a non-null value when
   exiting constructor. Consider adding the 'required' modifier or declaring the field as nullable.
  [Mobile] AAPL is now $150.75
  [Web] AAPL is now $150.75
  [Mobile] GOOGL is now $2825.3
  [Web] GOOGL is now $2825.3
○ PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp> 
```

Exercise 8: Implementing the Strategy Pattern

Scenario:

You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

```
using System;

public interface IPaymentStrategy
{
    void Pay(decimal amount);
}
```

```
public class CreditCardPayment : IPaymentStrategy
{
    public void Pay(decimal amount)
    {
        Console.WriteLine($"Paid {amount:C} using Credit Card.");
    }
}
```

```
public class PayPalPayment : IPaymentStrategy
{
    public void Pay(decimal amount)
    {
        Console.WriteLine($"Paid {amount:C} using PayPal.");
    }
}
```

```
public class PaymentContext
{
    private IPaymentStrategy _paymentStrategy;
```

```
    public void SetPaymentStrategy(IPaymentStrategy strategy)
    {
        _paymentStrategy = strategy;
    }
```

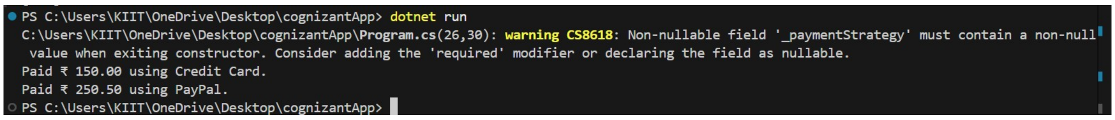
```
    public void ExecutePayment(decimal amount)
    {
        _paymentStrategy.Pay(amount);
    }
}
```

```
public class Program
{
    public static void Main(string[] args)
    {
        PaymentContext context = new PaymentContext();
```

```
        context.SetPaymentStrategy(new CreditCardPayment());
        context.ExecutePayment(150.00m);
```

```
        context.SetPaymentStrategy(new PayPalPayment());
        context.ExecutePayment(250.50m);
    }
}
```

Output:



```
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp> dotnet run
C:\Users\KIIT\OneDrive\Desktop\cognizantApp\Program.cs(26,30): warning CS8618: Non-nullable field '_paymentStrategy' must contain a non-null
 value when exiting constructor. Consider adding the 'required' modifier or declaring the field as nullable.
Paid ₹ 150.00 using Credit Card.
Paid ₹ 250.50 using PayPal.
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp>
```

Exercise 9: Implementing the Command Pattern

Scenario: You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

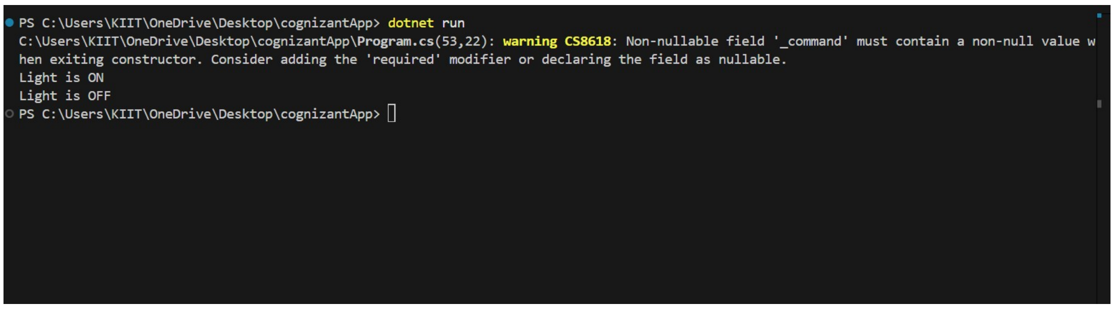
```
using System;

public interface ICommand
{
    void Execute();
}
```

```
public class Light
{
    public void TurnOn()
    {
        Console.WriteLine("Light is ON");
    }
```

```
    public void TurnOff()
    {
        Console.WriteLine("Light is OFF");
    }
}
```

```
public class LightOnCommand : ICommand
{
    private Light _light;
    public LightOnCommand(Light light)
    {
        _light = light;
    }

    public void Execute()
    {
        _light.TurnOn();
    }
}

public class LightOffCommand : ICommand
{
    private Light _light;

    public LightOffCommand(Light light)
    {
        _light = light;
    }

    public void Execute()
    {
        _light.TurnOff();
    }
}

public class RemoteControl
{
    private ICommand _command;

    public void SetCommand(ICommand command)
    {
        _command = command;
    }

    public void PressButton()
    {
        _command.Execute();
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        Light light = new Light();

        ICommand turnOn = new LightOnCommand(light);
        ICommand turnOff = new LightOffCommand(light);

        RemoteControl remote = new RemoteControl();

        remote.SetCommand(turnOn);
        remote.PressButton();

        remote.SetCommand(turnOff);
        remote.PressButton();
    }
}
```

Output:



```
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp> dotnet run
C:\Users\KIIT\OneDrive\Desktop\cognizantApp\Program.cs(53,22): warning CS8618: Non-nullable field '_command' must contain a non-null value w
hen exiting constructor. Consider adding the 'required' modifier or declaring the field as nullable.
Light is ON
Light is OFF
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp> 
```

Exercise 10: Implementing the MVC Pattern

Scenario:

You are developing a simple web application for managing student records using the MVC pattern.

```
using System;

public class Student
{
    public string Name { get; set; }
    public string Id { get; set; }
    public string Grade { get; set; }
}
```

```
public class StudentView
{
    public void DisplayStudentDetails(string name, string id, string grade)
    {
        Console.WriteLine($"Student: Name={name}, ID={id}, Grade={grade}");
    }
}
```

```
public class StudentController
{
    private Student _model;
    private StudentView _view;
```

```
    public StudentController(Student model, StudentView view)
    {
        _model = model;
        _view = view;
    }
```

```
    public void SetStudentName(string name)
    {
        _model.Name = name;
    }
```

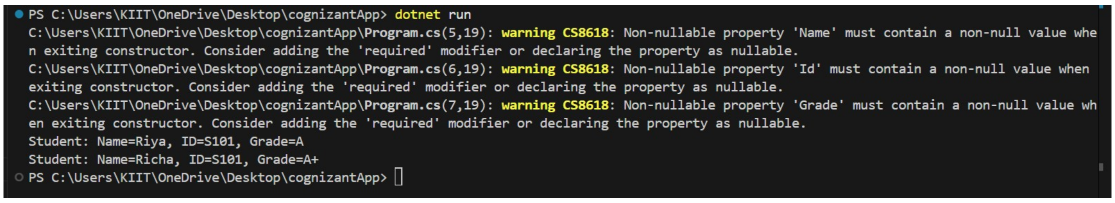
```
    public void SetStudentId(string id)
    {
        _model.Id = id;
    }
```

```
    public void SetStudentGrade(string grade)
    {
        _model.Grade = grade;
    }
```

```
    public void UpdateView()
    {
        _view.DisplayStudentDetails(_model.Name, _model.Id, _model.Grade);
    }
}
```

```
public class Program
{
    public static void Main(string[] args)
    {
        Student model = new Student { Name = "Riya", Id = "S101", Grade = "A" };
        StudentView view = new StudentView();
        StudentController controller = new StudentController(model, view);

        controller.UpdateView();

        controller.SetStudentName("Richa");
        controller.SetStudentGrade("A+");

        controller.UpdateView();
    }
}
```

Output:



```
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp> dotnet run
C:\Users\KIIT\OneDrive\Desktop\cognizantApp\Program.cs(5,19): warning CS8618: Non-nullable property 'Name' must contain a non-null value whe
n exiting constructor. Consider adding the 'required' modifier or declaring the property as nullable.
C:\Users\KIIT\OneDrive\Desktop\cognizantApp\Program.cs(6,19): warning CS8618: Non-nullable property 'Id' must contain a non-null value when
exiting constructor. Consider adding the 'required' modifier or declaring the property as nullable.
C:\Users\KIIT\OneDrive\Desktop\cognizantApp\Program.cs(7,19): warning CS8618: Non-nullable property 'Grade' must contain a non-null value wh
en exiting constructor. Consider adding the 'required' modifier or declaring the property as nullable.
Student: Name=Riya, ID=S101, Grade=A
Student: Name=Richa, ID=S101, Grade=A+
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp>
```

Exercise 11: Implementing Dependency Injection

Scenario:

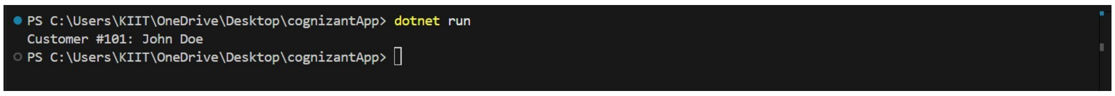
You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

```
using System;

public interface ICustomerRepository
{
    string FindCustomerById(int id);
}

public class CustomerRepositoryImpl : ICustomerRepository
{
    public string FindCustomerById(int id)
    {
        return $"Customer #{id}: John Doe";
    }
}

public class CustomerService
{
    private readonly ICustomerRepository _repository;

    public CustomerService(ICustomerRepository repository)
    {
        _repository = repository;
    }

    public void GetCustomerDetails(int id)
    {
        string customer = _repository.FindCustomerById(id);
        Console.WriteLine(customer);
    }
}

public class Program
```

```
{
    public static void Main(string[] args)
    {
        ICustomerRepository repo = new CustomerRepositoryImpl();
        CustomerService service = new CustomerService(repo);
```

```
        service.GetCustomerDetails(101);
    }
}
```

Output:



```
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp> dotnet run
Customer #101: John Doe
PS C:\Users\KIIT\OneDrive\Desktop\cognizantApp>
```