**Exercise 1: Implementing the Singleton Pattern**

Codes :

Logger.cs:

public class Logger

{

    private static Logger \_instance;

    private static readonly object \_lock = new object();

    private Logger()

    {

        Console.WriteLine("Logger instance created.");

    }

    public static Logger GetInstance()

    {

        if (\_instance == null)

        {

            lock (\_lock)

            {

                if (\_instance == null)

                {

                    \_instance = new Logger();

                }

            }

        }

        return \_instance;

    }

    public void Log(string message)

    {

        Console.WriteLine($"[Log]: {message}");

    }

}

Program.cs:

using System;

class Program

{

    static void Main(string[] args)

    {

        Logger logger1 = Logger.GetInstance();

        logger1.Log("First log message.");

        Logger logger2 = Logger.GetInstance();

        logger2.Log("Second log message.");

        if (logger1 == logger2)

        {

            Console.WriteLine("Same Logger instance is used.");

        }

        else

        {

            Console.WriteLine("Different Logger instances exist.");

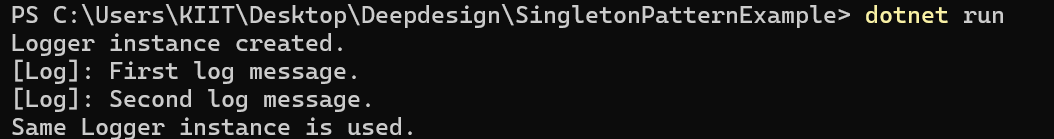
        }

        Console.ReadLine();

    }

}

Output:



**Exercise 2: Implementing the Factory Method Pattern**

Codes :

IDocment.cs:

public interface IDocument

{

    void Open();

}

WordDocment.cs:

public class WordDocument : IDocument

{

    public void Open()

    {

        Console.WriteLine("Opening a Word document.");

    }

}

PdfDocment.cs:

public class PdfDocument : IDocument

{

    public void Open()

    {

        Console.WriteLine("Opening a PDF document.");

    }

}

ExcelDocment.cs:

public class ExcelDocument : IDocument

{

    public void Open()

    {

        Console.WriteLine("Opening an Excel document.");

    }

}

DocumentFactory.cs:

public abstract class DocumentFactory

{

    public abstract IDocument CreateDocument();

}

WordDocumentFactory.cs:

public class WordDocumentFactory : DocumentFactory

{

    public override IDocument CreateDocument()

    {

        return new WordDocument();

    }

}

PdfDocumentFactory.cs:

public class PdfDocumentFactory : DocumentFactory

{

    public override IDocument CreateDocument()

    {

        return new PdfDocument();

    }

}

ExcelDocumentFactory.cs:

public class ExcelDocumentFactory : DocumentFactory

{

    public override IDocument CreateDocument()

    {

        return new ExcelDocument();

    }

}

Program.cs:

using System;

class Program

{

    static void Main(string[] args)

    {

        DocumentFactory wordFactory = new WordDocumentFactory();

        IDocument wordDoc = wordFactory.CreateDocument();

        wordDoc.Open();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        IDocument pdfDoc = pdfFactory.CreateDocument();

        pdfDoc.Open();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

        IDocument excelDoc = excelFactory.CreateDocument();

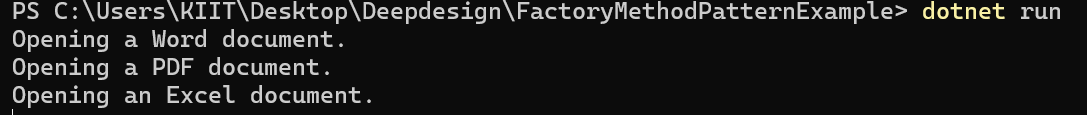
        excelDoc.Open();

        Console.ReadLine();

    }

}

Output:



**Exercise 3: Implementing the Builder Pattern**

Codes :

Computer.cs:

public class Computer

{

    public string CPU { get; }

    public string RAM { get; }

    public string Storage { get; }

    public string GraphicsCard { get; }

    private Computer(Builder builder)

    {

        CPU = builder.CPU;

        RAM = builder.RAM;

        Storage = builder.Storage;

        GraphicsCard = builder.GraphicsCard;

    }

    public class Builder

    {

        public string CPU { get; private set; }

        public string RAM { get; private set; }

        public string Storage { get; private set; }

        public string GraphicsCard { get; private set; }

        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 Builder SetGraphicsCard(string graphicsCard)

        {

            GraphicsCard = graphicsCard;

            return this;

        }

        public Computer Build()

        {

            return new Computer(this);

        }

    }

    public void ShowConfiguration()

    {

        Console.WriteLine($"CPU: {CPU}, RAM: {RAM}, Storage: {Storage}, Graphics: {GraphicsCard}");

    }

}

Program.cs:

using System;

class Program

{

    static void Main(string[] args)

    {

        var gamingComputer = new Computer.Builder()

            .SetCPU("Intel i9")

            .SetRAM("32GB")

            .SetStorage("1TB SSD")

            .SetGraphicsCard("NVIDIA RTX 4080")

            .Build();

             var officeComputer = new Computer.Builder()

            .SetCPU("Intel i3")

            .SetRAM("8GB")

            .SetStorage("500GB HDD")

            .Build();

        Console.WriteLine("Gaming Computer Configuration:");

        gamingComputer.ShowConfiguration();

        Console.WriteLine("\nOffice Computer Configuration:");

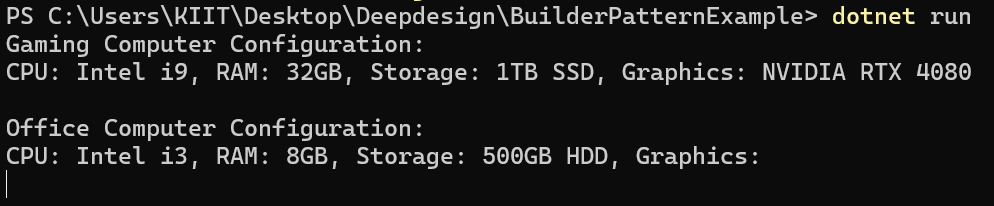
        officeComputer.ShowConfiguration();

        Console.ReadLine();

    }

}

Output:



**Exercise 4: Implementing the Adapter Pattern**

Codes :

IPaymentProcessor.cs:

public interface IPaymentProcessor

{

    void ProcessPayment(decimal amount);

}

PayPalGateway.cs:

public class PayPalGateway

{

    public void MakePayment(double usdAmount)

    {

        Console.WriteLine($"[PayPal] Paid ${usdAmount} successfully.");

    }

}

StripeGateway.cs:

public class StripeGateway

{

    public void Charge(decimal rupeeAmount)

    {

        Console.WriteLine($"[Stripe] Charged ₹{rupeeAmount} successfully.");

    }

}

PayPalAdapter.cs:

public class PayPalAdapter : IPaymentProcessor

{

    private readonly PayPalGateway \_paypal;

    public PayPalAdapter(PayPalGateway paypal)

    {

        \_paypal = paypal;

    }

    public void ProcessPayment(decimal amount)

    {

        \_paypal.MakePayment((double)amount);

    }

}

StripeAdapter.cs:

public class StripeAdapter : IPaymentProcessor

{

    private readonly StripeGateway \_stripe;

    public StripeAdapter(StripeGateway stripe)

    {

        \_stripe = stripe;

    }

    public void ProcessPayment(decimal amount)

    {

        \_stripe.Charge(amount);

    }

}

Program.cs:

using System;

class Program

{

     static void Main(string[] args)

    {

        IPaymentProcessor paypalProcessor = new PayPalAdapter(new PayPalGateway());

        paypalProcessor.ProcessPayment(500.00m);

        IPaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());

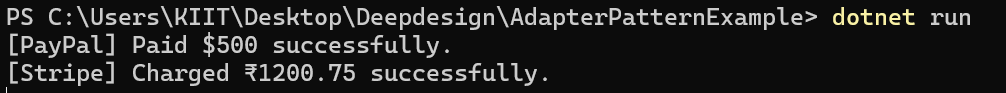
        stripeProcessor.ProcessPayment(1200.75m);

        Console.ReadLine();

    }

}

Output:



**Exercise 5: Implementing the Decorator Pattern**

Codes :

INotifier.cs:

public interface INotifier

{

    void Send(string message);

}

EmailNotifier.cs:

public class EmailNotifier : INotifier

{

    public void Send(string message)

    {

        Console.WriteLine($"Sending Email: {message}");

    }

}

NotifierDecorator.cs:

public abstract class NotifierDecorator : INotifier

{

    protected INotifier \_wrappedNotifier;

    protected NotifierDecorator(INotifier notifier)

    {

        \_wrappedNotifier = notifier;

    }

    public virtual void Send(string message)

    {

        \_wrappedNotifier.Send(message);

    }

}

SMSNotifierDecorator.cs:

public class SMSNotifierDecorator : NotifierDecorator

{

    public SMSNotifierDecorator(INotifier notifier) : base(notifier) { }

    public override void Send(string message)

    {

        base.Send(message);

        Console.WriteLine($"Sending SMS: {message}");

    }

}

SlackNotifierDecorator.cs:

public class SlackNotifierDecorator : NotifierDecorator

{

    public SlackNotifierDecorator(INotifier notifier) : base(notifier) { }

    public override void Send(string message)

    {

        base.Send(message);

        Console.WriteLine($"Sending Slack message: {message}");

    }

}

Program.cs:

using System;

class Program

{

    static void Main(string[] args)

    {

        INotifier notifier = new EmailNotifier();

        notifier = new SMSNotifierDecorator(notifier);

        notifier = new SlackNotifierDecorator(notifier);

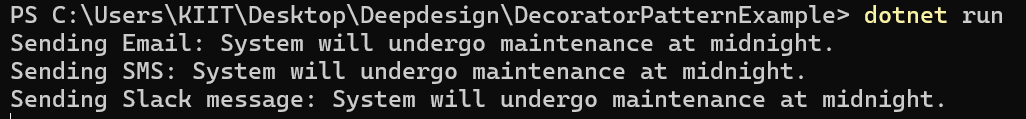
         notifier.Send("System will undergo maintenance at midnight.");

        Console.ReadLine();

    }

}

Output:



**Exercise 6: Implementing the Proxy Pattern**

Codes :

IImage.cs:

public interface IImage

{

    void Display();

}

RealImage.cs:

using System;

public class RealImage : IImage

{

     private string \_fileName;

    public RealImage(string fileName)

    {

        \_fileName = fileName;

        LoadFromServer(fileName);

    }

    private void LoadFromServer(string fileName)

    {

        Console.WriteLine($"Loading image '{fileName}' from remote server...");

    }

     public void Display()

    {

        Console.WriteLine($"Displaying image: {\_fileName}");

    }

}

ProxyImage.cs:

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); // Lazy loading

        }

        else

        {

            Console.WriteLine($"Using cached image: {\_fileName}");

        }

        \_realImage.Display();

    }

}

Program.cs:

using System;

class Program

{

    static void Main(string[] args)

    {

        IImage image1 = new ProxyImage("photo1.jpg");

        IImage image2 = new ProxyImage("photo2.jpg");

        Console.WriteLine("\nFirst call to image1:");

        image1.Display();

        Console.WriteLine("\nSecond call to image1:");

        image1.Display();

        Console.WriteLine("\nFirst call to image2:");

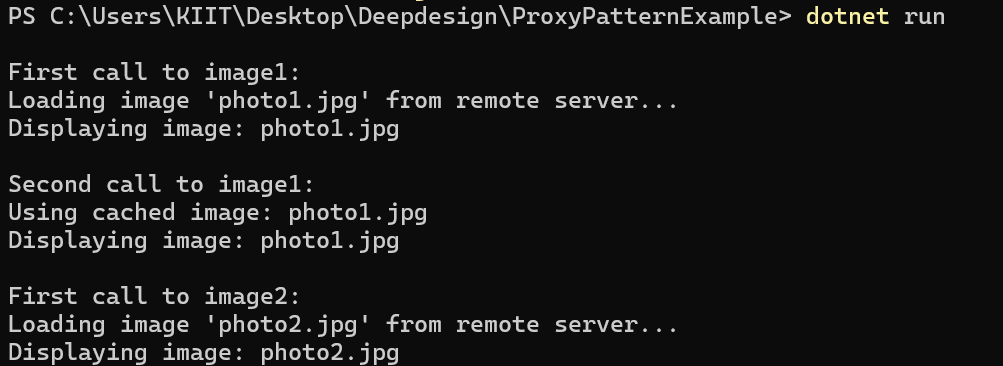
        image2.Display();

        Console.ReadLine();

    }

}

Output:



**Exercise 7: Implementing the Observer Pattern**

Codes :

IStock.cs:

using System;

public interface IStock

{

    void RegisterObserver(IObserver observer);

    void RemoveObserver(IObserver observer);

    void NotifyObservers();

}

StockMarket.cs:

using System;

using System.Collections.Generic;

public class StockMarket : IStock

{

    private readonly List<IObserver> \_observers = new();

    private string \_stockName;

    private decimal \_stockPrice;

    public StockMarket(string stockName)

    {

        \_stockName = stockName;

    }

    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(\_stockName, \_stockPrice);

        }

    }

    public void SetPrice(decimal newPrice)

    {

        Console.WriteLine($"\n[StockMarket] {\_stockName} price updated to ₹{newPrice}");

        \_stockPrice = newPrice;

        NotifyObservers();

    }

}

IObserver.cs:

public interface IObserver

{

    void Update(string stockName, decimal stockPrice);

}

MobileApp.cs:

using System;

public class MobileApp : IObserver

{

    private readonly string \_user;

    public MobileApp(string user)

    {

        \_user = user;

    }

    public void Update(string stockName, decimal stockPrice)

    {

        Console.WriteLine($"[MobileApp - {\_user}] {stockName} is now ₹{stockPrice}");

    }

}

WebApp.cs:

using System;

public class WebApp : IObserver

{

    private readonly string \_user;

    public WebApp(string user)

    {

        \_user = user;

    }

    public void Update(string stockName, decimal stockPrice)

    {

        Console.WriteLine($"[WebApp - {\_user}] {stockName} is now ₹{stockPrice}");

    }

}

Program.cs:

using System;

class Program

{

    static void Main(string[] args)

    {

        StockMarket tcsStock = new StockMarket("TCS");

        IObserver mobileUser = new MobileApp("Alice");

        IObserver webUser = new WebApp("Bob");

        tcsStock.RegisterObserver(mobileUser);

        tcsStock.RegisterObserver(webUser);

        tcsStock.SetPrice(3500.50m);

        tcsStock.SetPrice(3600.00m);

        tcsStock.RemoveObserver(mobileUser);

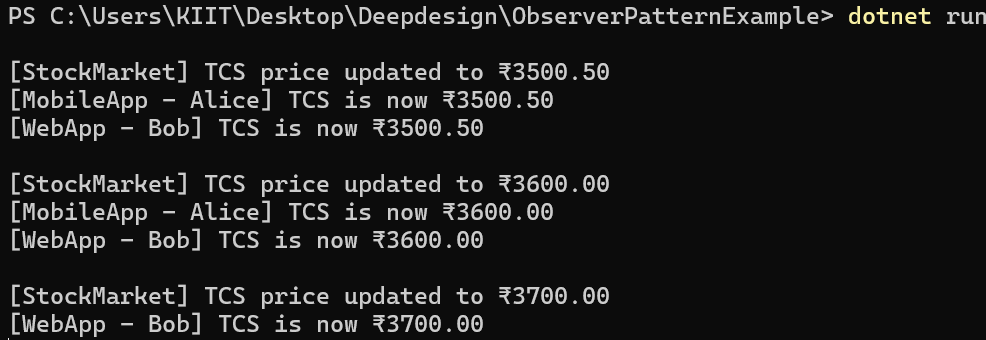
        tcsStock.SetPrice(3700.00m);

        Console.ReadLine();

    }

}

Output:



**Exercise 8: Implementing the Strategy Pattern**

Codes :

IPaymentStrategy.cs:

public interface IPaymentStrategy

{

    void Pay(decimal amount);

}

CreditCardPayment.cs:

using System;

public class CreditCardPayment : IPaymentStrategy

{

     private string \_cardNumber;

    public CreditCardPayment(string cardNumber)

    {

        \_cardNumber = cardNumber;

    }

    public void Pay(decimal amount)

    {

        Console.WriteLine($"Paid ₹{amount} using Credit Card: {\_cardNumber}");

    }

}

PayPalPayment.cs:

using System;

public class PayPalPayment : IPaymentStrategy

{

    private string \_email;

    public PayPalPayment(string email)

    {

        \_email = email;

    }

    public void Pay(decimal amount)

    {

        Console.WriteLine($"Paid ₹{amount} using PayPal: {\_email}");

    }

}

PaymentContext.cs:

public class PaymentContext

{

    private IPaymentStrategy \_paymentStrategy;

    public void SetPaymentStrategy(IPaymentStrategy paymentStrategy)

    {

        \_paymentStrategy = paymentStrategy;

    }

    public void Pay(decimal amount)

    {

        if (\_paymentStrategy == null)

        {

            Console.WriteLine("Payment strategy is not set.");

            return;

        }

        \_paymentStrategy.Pay(amount);

    }

}

Program.cs:

using System;

class Program

{

    static void Main(string[] args)

    {

        PaymentContext context = new PaymentContext();

        context.SetPaymentStrategy(new CreditCardPayment("1234-5678-9876-5432"));

        context.Pay(1500.00m);

        context.SetPaymentStrategy(new PayPalPayment("user@example.com"));

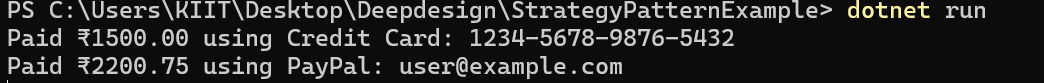
        context.Pay(2200.75m);

        Console.ReadLine();

    }

}

Output:



**Exercise 9: Implementing the Command Pattern**

Codes :

ICommand.cs:

public interface ICommand

{

    void Execute();

}

Light.cs:

using System;

public class Light

{

    public void TurnOn()

    {

        Console.WriteLine("Light is ON");

    }

    public void TurnOff()

    {

        Console.WriteLine("Light is OFF");

    }

}

LightOnCommand.cs:

public class LightOnCommand : ICommand

{

     private readonly Light \_light;

    public LightOnCommand(Light light)

    {

        \_light = light;

    }

    public void Execute()

    {

        \_light.TurnOn();

    }

}

LightOffCommand.cs:

public class LightOffCommand : ICommand

{

    private readonly Light \_light;

    public LightOffCommand(Light light)

    {

        \_light = light;

    }

    public void Execute()

    {

        \_light.TurnOff();

    }

}

RemoteControl.cs:

public class RemoteControl

{

    private ICommand \_command;

    public void SetCommand(ICommand command)

    {

        \_command = command;

    }

    public void PressButton()

    {

        \_command.Execute();

    }

}

Program.cs:

using System;

class Program

{

    static void Main(string[] args)

    {

         Light livingRoomLight = new Light();

        ICommand lightOn = new LightOnCommand(livingRoomLight);

        ICommand lightOff = new LightOffCommand(livingRoomLight);

        RemoteControl remote = new RemoteControl();

        remote.SetCommand(lightOn);

        remote.PressButton();

        remote.SetCommand(lightOff);

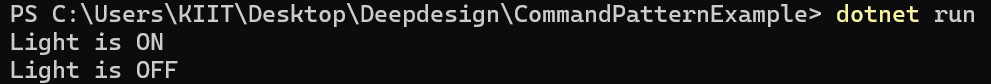
        remote.PressButton();

         Console.ReadLine();

    }

}

Output:



**Exercise 10: Implementing the MVC Pattern**

Codes :

Student.cs:

public class Student

{

    public string Name { get; set; }

    public string Id { get; set; }

    public string Grade { get; set; }

}

StudentView.cs:

using System;

public class StudentView

{

    public void DisplayStudentDetails(string studentName, string studentId, string studentGrade)

    {

        Console.WriteLine("Student Details:");

        Console.WriteLine($"Name  : {studentName}");

        Console.WriteLine($"ID    : {studentId}");

        Console.WriteLine($"Grade : {studentGrade}");

        Console.WriteLine();

    }

}

StudentController.cs:

public class StudentController

{

    private readonly Student \_model;

    private readonly 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 string GetStudentName() => \_model.Name;

    public string GetStudentId() => \_model.Id;

    public string GetStudentGrade() => \_model.Grade;

    public void UpdateView()

    {

        \_view.DisplayStudentDetails(\_model.Name, \_model.Id, \_model.Grade);

    }

}

Program.cs:

using System;

class Program

{

    static void Main(string[] args)

    {

        Student student = new Student

        {

            Name = "Alice",

            Id = "S101",

            Grade = "A"

        };

        StudentView view = new StudentView();

        StudentController controller = new StudentController(student, view);

        controller.UpdateView();

        controller.SetStudentName("Alice Johnson");

        controller.SetStudentGrade("A+");

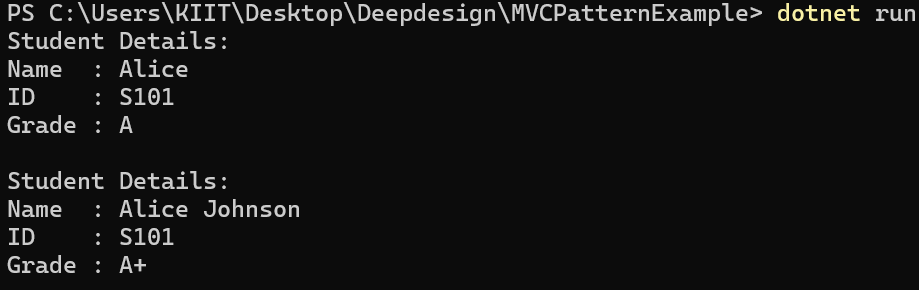
        controller.UpdateView();

        Console.ReadLine();

    }

}

Output:



**Exercise 11: Implementing Dependency Injection**

Codes :

ICustomerRepository.cs:

public interface ICustomerRepository

{

    string FindCustomerById(int id);

}

CustomerRepositoryImpl.cs:

using System;

using System.Collections.Generic;

public class CustomerRepositoryImpl : ICustomerRepository

{

    private readonly Dictionary<int, string> \_customers = new()

    {

        { 1, "Alice Johnson" },

        { 2, "Bob Smith" },

        { 3, "Charlie Davis" }

    };

    public string FindCustomerById(int id)

    {

        return \_customers.ContainsKey(id)

            ? \_customers[id]

            : "Customer not found";

    }

}

CustomerService.cs

using System;

public class CustomerService

{

    private readonly ICustomerRepository \_repository;

    public CustomerService(ICustomerRepository repository)

    {

        \_repository = repository;

    }

    public void PrintCustomer(int id)

    {

        string customer = \_repository.FindCustomerById(id);

        Console.WriteLine($"Customer (ID: {id}) -> {customer}");

    }

}

Program.cs

using System;

class Program

{

    static void Main(string[] args)

    {

        ICustomerRepository repository = new CustomerRepositoryImpl();

        CustomerService service = new CustomerService(repository);

        service.PrintCustomer(1);

        service.PrintCustomer(2);

        service.PrintCustomer(5);

        Console.ReadLine();

    }

}

Output:

