**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.  
  
Solution :  
  
Program :

using System;

public class Logger

{

private static Logger \_instance;

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

private Logger()

{

Console.WriteLine("Logger initialized.");

}

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);

}

}

class Program

{

static void Main(string[] args)

{

Logger logger1 = Logger.GetInstance();

logger1.Log("This is the first message.");

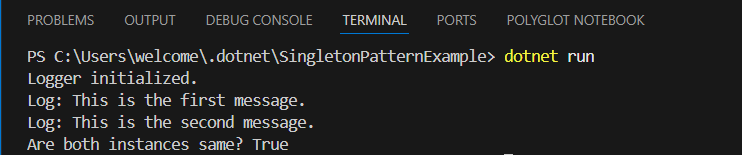
Logger logger2 = Logger.GetInstance();

logger2.Log("This is the second message.");

Console.WriteLine($"Are both instances same? {ReferenceEquals(logger1, logger2)}");

}

**}  
  
  
  
Output :**



**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.

**Solution :  
  
Program :**

using System;

public interface IDocument

{

void Open();

}

public class WordDocument : IDocument

{

public void Open()

{

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

}

}

public class PdfDocument : IDocument

{

public void Open()

{

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

}

}

public class ExcelDocument : IDocument

{

public void Open()

{

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

}

}

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

}

public class WordDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new WordDocument();

}

}

public class PdfDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new PdfDocument();

}

}

public class ExcelDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new ExcelDocument();

}

}

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();

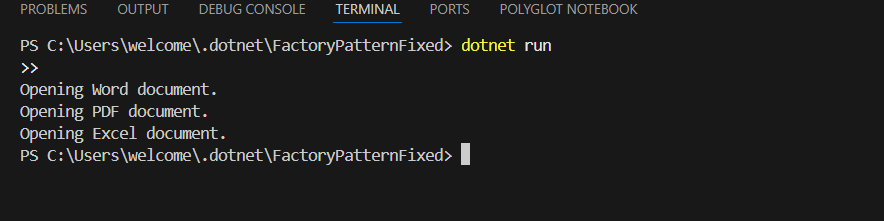
IDocument excelDoc = excelFactory.CreateDocument();

excelDoc.Open();

}

}

**Output** :



**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.

**Solution** :

**Program :**

using System;

class Computer

{

public string CPU { get; private set; }

public string RAM { get; private set; }

public string Storage { get; private set; }

public string GraphicsCard { get; private set; }

private Computer(Builder builder)

{

CPU = builder.CPU;

RAM = builder.RAM;

Storage = builder.Storage;

GraphicsCard = builder.GraphicsCard;

}

public override string ToString()

{

return $"CPU: {CPU}, RAM: {RAM}, Storage: {Storage}, Graphics Card: {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);

}

}

}

class Program

{

public static void Main(string[] args)

{

var gamingPC = new Computer.Builder()

.SetCPU("Intel Core i9")

.SetRAM("32GB")

.SetStorage("1TB SSD")

.SetGraphicsCard("NVIDIA RTX 4090")

.Build();

var officePC = new Computer.Builder()

.SetCPU("Intel Core i5")

.SetRAM("16GB")

.SetStorage("512GB SSD")

.Build(); // No graphics card

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

Console.WriteLine(gamingPC);

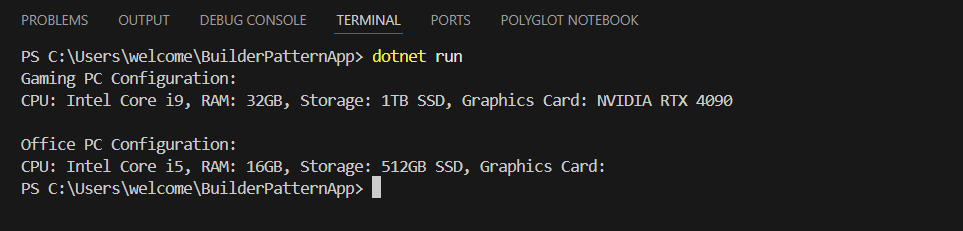
Console.WriteLine("\nOffice PC Configuration:");

Console.WriteLine(officePC);

}

}

**Output :**



**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.

**Solution :**

**Program :**

using System;

public interface IPaymentProcessor

{

void ProcessPayment(string customerName, double amount);

}

public class PayPalGateway

{

public void SendPayment(string user, double money)

{

Console.WriteLine($"PayPal: Processed payment of ${money} for {user}.");

}

}

public class StripeGateway

{

public void MakePayment(string client, double amount)

{

Console.WriteLine($"Stripe: Processed payment of ${amount} for {client}.");

}

}

public class PayPalAdapter : IPaymentProcessor

{

private PayPalGateway \_paypal = new PayPalGateway();

public void ProcessPayment(string customerName, double amount)

{

\_paypal.SendPayment(customerName, amount);

}

}

public class StripeAdapter : IPaymentProcessor

{

private StripeGateway \_stripe = new StripeGateway();

public void ProcessPayment(string customerName, double amount)

{

\_stripe.MakePayment(customerName, amount);

}

}

class Program

{

static void Main(string[] args)

{

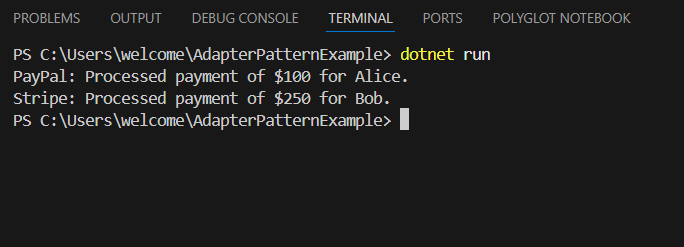
IPaymentProcessor paypal = new PayPalAdapter();

paypal.ProcessPayment("Alice", 100.00);

IPaymentProcessor stripe = new StripeAdapter();

stripe.ProcessPayment("Bob", 250.00);

}

}  
  
  
**Output** :  


**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.

**Solution :**

**Program :**

using System;

public interface INotifier

{

void Send(string message);

}

public class EmailNotifier : INotifier

{

public void Send(string message)

{

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

}

}

public abstract class NotifierDecorator : INotifier

{

protected INotifier \_notifier;

protected NotifierDecorator(INotifier notifier)

{

\_notifier = notifier;

}

public virtual void Send(string message)

{

\_notifier.Send(message);

}

}

public class SMSNotifierDecorator : NotifierDecorator

{

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

public override void Send(string message)

{

base.Send(message);

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

}

}

public class SlackNotifierDecorator : NotifierDecorator

{

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

public override void Send(string message)

{

base.Send(message);

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

}

}

class Program

{

static void Main(string[] args)

{

INotifier notifier = new EmailNotifier();

notifier = new SMSNotifierDecorator(notifier);

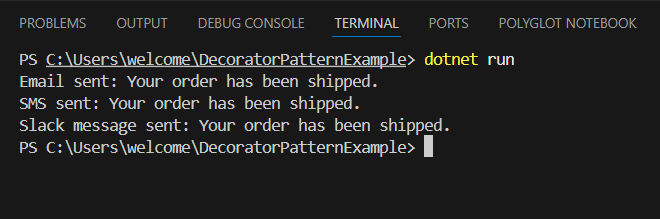
notifier = new SlackNotifierDecorator(notifier);

notifier.Send("Your order has been shipped.");

}

}

**Output** :



**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.

**Solution :**

**Program :**

using System;

namespace ProxyPatternExample

{

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();

}

}

class Program

{

static void Main(string[] args)

{

IImage image = new ProxyImage("photo.jpg");

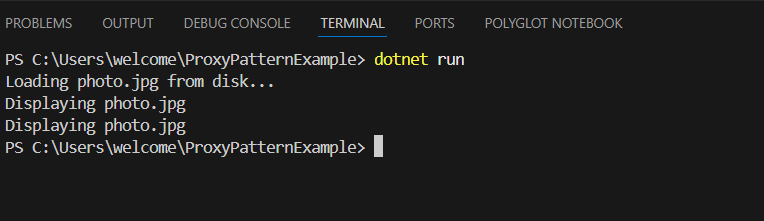
image.Display();

image.Display();

}

}

}  
 **Output :**



**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.**

**Solution :**

**Program :**

using System;

using System.Collections.Generic;

// Step 4: Observer Interface

public interface IObserver

{

void Update(string stockName, double newPrice);

}

// Step 2: Subject Interface

public interface IStock

{

void RegisterObserver(IObserver observer);

void DeregisterObserver(IObserver observer);

void NotifyObservers();

}

// Step 3: Concrete Subject

public class StockMarket : IStock

{

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

private string \_stockName;

private double \_stockPrice;

public StockMarket(string stockName, double initialPrice)

{

\_stockName = stockName;

\_stockPrice = initialPrice;

}

public void RegisterObserver(IObserver observer)

{

\_observers.Add(observer);

}

public void DeregisterObserver(IObserver observer)

{

\_observers.Remove(observer);

}

public void NotifyObservers()

{

foreach (var observer in \_observers)

{

observer.Update(\_stockName, \_stockPrice);

}

}

public void SetPrice(double newPrice)

{

Console.WriteLine($"\nStock price updated: {\_stockName} - ${newPrice}");

\_stockPrice = newPrice;

NotifyObservers();

}

}

// Step 5: Concrete Observers

public class MobileApp : IObserver

{

private string \_appName;

public MobileApp(string appName)

{

\_appName = appName;

}

public void Update(string stockName, double newPrice)

{

Console.WriteLine($"{\_appName} App: {stockName} is now ${newPrice}");

}

}

public class WebApp : IObserver

{

private string \_appName;

public WebApp(string appName)

{

\_appName = appName;

}

public void Update(string stockName, double newPrice)

{

Console.WriteLine($"{\_appName} WebApp: {stockName} is now ${newPrice}");

}

}

// Step 6: Test the Observer Pattern

class Program

{

static void Main(string[] args)

{

StockMarket appleStock = new StockMarket("AAPL", 150.00);

IObserver mobile = new MobileApp("StockViewer");

IObserver web = new WebApp("MarketWatch");

appleStock.RegisterObserver(mobile);

appleStock.RegisterObserver(web);

appleStock.SetPrice(152.75);

appleStock.SetPrice(149.20);

// Deregister one observer

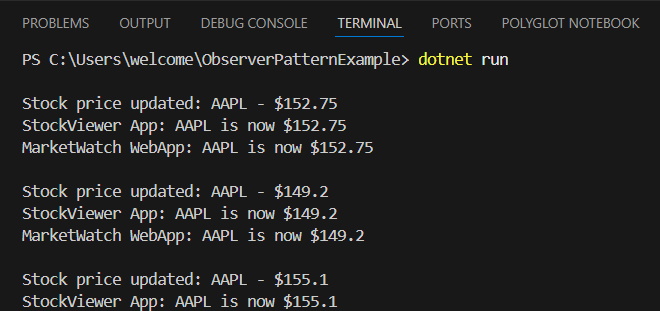
appleStock.DeregisterObserver(web);

appleStock.SetPrice(155.10);

}

}

**Output :**



**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.**

**Solution :**

**Program :**

using System;

// Step 2: Strategy Interface

public interface IPaymentStrategy

{

void Pay(double amount);

}

// Step 3: Concrete Strategies

public class CreditCardPayment : IPaymentStrategy

{

private string \_cardNumber;

public CreditCardPayment(string cardNumber)

{

\_cardNumber = cardNumber;

}

public void Pay(double amount)

{

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

}

}

public class PayPalPayment : IPaymentStrategy

{

private string \_email;

public PayPalPayment(string email)

{

\_email = email;

}

public void Pay(double amount)

{

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

}

}

// Step 4: Context Class

public class PaymentContext

{

private IPaymentStrategy \_paymentStrategy;

public PaymentContext(IPaymentStrategy paymentStrategy)

{

\_paymentStrategy = paymentStrategy;

}

public void ExecutePayment(double amount)

{

\_paymentStrategy.Pay(amount);

}

public void SetStrategy(IPaymentStrategy strategy)

{

\_paymentStrategy = strategy;

}

}

// Step 5: Test Class

class Program

{

static void Main(string[] args)

{

// Using Credit Card payment

IPaymentStrategy creditCard = new CreditCardPayment("1234-5678-9012-3456");

PaymentContext context = new PaymentContext(creditCard);

context.ExecutePayment(250.00);

// Switching to PayPal payment

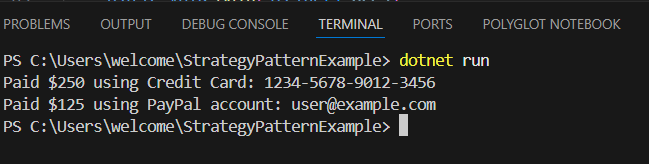
IPaymentStrategy paypal = new PayPalPayment("user@example.com");

context.SetStrategy(paypal);

context.ExecutePayment(125.00);

}

}  
  
**Output :**



**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.

**Solution :**

**Program :**

using System;

// Step 2: Command Interface

public interface ICommand

{

    void Execute();

}

// Step 5: Receiver Class

public class Light

{

    public void TurnOn()

    {

        Console.WriteLine("Light is ON");

    }

    public void TurnOff()

    {

        Console.WriteLine("Light is OFF");

    }

}

// Step 3: Concrete Commands

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();

    }

}

// Step 4: Invoker Class

public class RemoteControl

{

    private ICommand? \_command;

    public void SetCommand(ICommand command)

    {

        \_command = command;

    }

    public void PressButton()

    {

        \_command?.Execute();

    }

}

// Step 6: Test the Command Pattern

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();

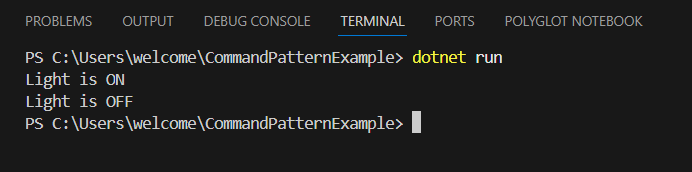
        remote.SetCommand(lightOff);

        remote.PressButton();

    }

}

Output :



**Exercise 10: Implementing the MVC Pattern**

**Scenario:**

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

**Solution :**

**Program :**

using System;

// Step 2: Model

public class Student

{

    public string Name { get; set; }

    public string Id { get; set; }

    public string Grade { get; set; }

    public Student(string name, string id, string grade)

    {

        Name = name;

        Id = id;

        Grade = grade;

    }

}

// Step 3: View

public class StudentView

{

    public void DisplayStudentDetails(string name, string id, string grade)

    {

        Console.WriteLine("Student Details:");

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

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

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

    }

}

// Step 4: Controller

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 string GetStudentName() => \_model.Name;

    public void SetStudentId(string id) => \_model.Id = id;

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

    public void SetStudentGrade(string grade) => \_model.Grade = grade;

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

    public void UpdateView()

    {

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

    }

}

// Step 5: Test the MVC Pattern

class Program

{

    static void Main(string[] args)

    {

        // Create model

        Student model = new Student("Alice", "S101", "A");

        // Create view

        StudentView view = new StudentView();

        // Create controller

        StudentController controller = new StudentController(model, view);

        // Display initial data

        controller.UpdateView();

        // Update model data using controller

        controller.SetStudentName("Bob");

        controller.SetStudentGrade("B+");

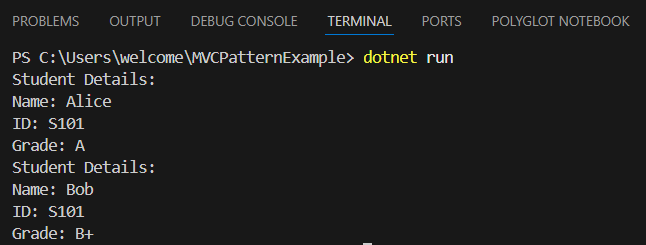
        // Display updated data

        controller.UpdateView();

    }

}

**Output :**



**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.**

**Solution :**

**Program :**

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 \_customerRepository;

public CustomerService(ICustomerRepository customerRepository)

{

\_customerRepository = customerRepository;

}

public void DisplayCustomer(int id)

{

string customer = \_customerRepository.FindCustomerById(id);

Console.WriteLine(customer);

}

}

class Program

{

static void Main(string[] args)

{

ICustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

service.DisplayCustomer(101);

}

}

**Output :**

