**Exercise 1: Implementing the Singleton Pattern**

**Solution**

**Filename - Logger.cs**

using System;

namespace SingletonPatternExample

{

    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 entry: {message}");

        }

    }

}

**Filename – Program.cs**

using System;

namespace SingletonPatternExample

{

    class Program

    {

        static void Main(string[] args)

        {

            Logger logger1 = Logger.GetInstance();

            Logger logger2 = Logger.GetInstance();

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

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

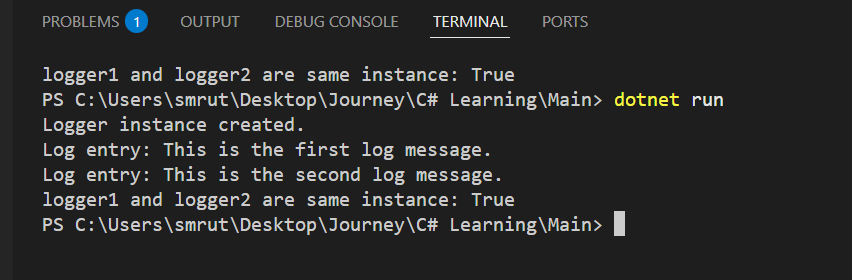
            Console.WriteLine($"logger1 and logger2 are same instance: {object.ReferenceEquals(logger1, logger2)}");

        }

    }

}

**OUTPUT**

****

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

**Solution**

**Filename – Exercise\_2.cs**

using System;

namespace FactoryMethodPatternExample

{

    // Document interface

    public interface IDocument

    {

        void Open();

    }

    // Concrete document types

    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...");

    }

    // Abstract factory

    public abstract class DocumentFactory

    {

        public abstract IDocument CreateDocument();

    }

    // Concrete factories

    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();

    }

    // Main class

    class Program

    {

        static void Main(string[] args)

        {

            DocumentFactory wordFactory = new WordFactory();

            IDocument word = wordFactory.CreateDocument();

            word.Open();

            DocumentFactory pdfFactory = new PdfFactory();

            IDocument pdf = pdfFactory.CreateDocument();

            pdf.Open();

            DocumentFactory excelFactory = new ExcelFactory();

            IDocument excel = excelFactory.CreateDocument();

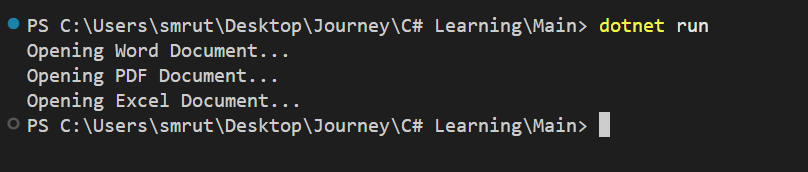
            excel.Open();

        }

    }

}

**OUTPUT**



**Exercise 3: Implementing the Builder Pattern**

**Solution**

**Filename – Exercise\_3.cs**

using System;

namespace BuilderPatternExample

{

    public class Computer

    {

        public string CPU { get; }

        public string RAM { get; }

        public string Storage { get; }

        public string GPU { get; }

        private Computer(Builder builder)

        {

            CPU = builder.CPU;

            RAM = builder.RAM;

            Storage = builder.Storage;

            GPU = builder.GPU;

        }

        public class Builder

        {

            public string CPU;

            public string RAM;

            public string Storage;

            public string GPU;

            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 SetGPU(string gpu)

            {

                GPU = gpu;

                return this;

            }

            public Computer Build()

            {

                return new Computer(this);

            }

        }

        public void DisplayConfig()

        {

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

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            var gamingPC = new Computer.Builder()

                .SetCPU("Intel i9")

                .SetRAM("32GB")

                .SetStorage("1TB SSD")

                .SetGPU("NVIDIA RTX 4090")

                .Build();

            var officePC = new Computer.Builder()

                .SetCPU("Intel i5")

                .SetRAM("8GB")

                .SetStorage("512GB SSD")

                .Build(); // No GPU

            gamingPC.DisplayConfig();

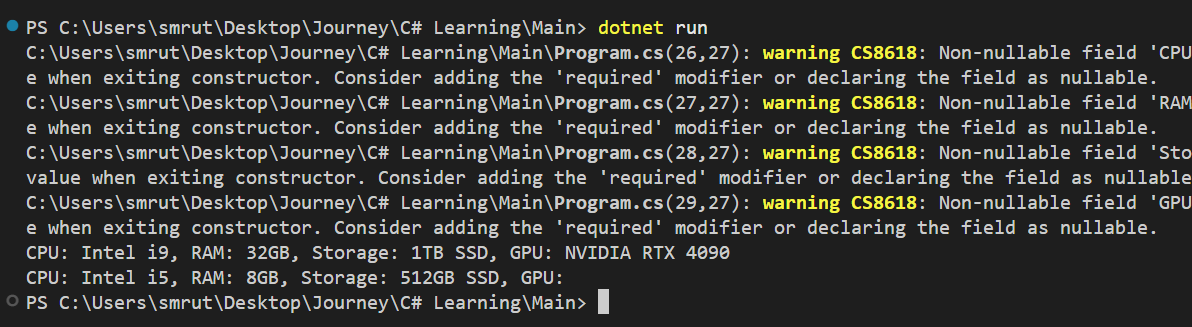
            officePC.DisplayConfig();

        }

    }

}

**OUTPUT**



**Exercise 4: Implementing the Adapter Pattern**

**Solution**

**Filename – Exercise\_4.cs**

using System;

namespace AdapterPatternExample

{

    public interface IPaymentProcessor

    {

        void ProcessPayment(string amount);

    }

    public class PayPalGateway

    {

        public void MakePayment(string money)

        {

            Console.WriteLine($"[PayPal] Payment of {money} processed.");

        }

    }

    public class StripeGateway

    {

        public void SendPayment(string cash)

        {

            Console.WriteLine($"[Stripe] Payment of {cash} processed.");

        }

    }

    public class PayPalAdapter : IPaymentProcessor

    {

        private PayPalGateway \_paypal = new PayPalGateway();

        public void ProcessPayment(string amount)

        {

            \_paypal.MakePayment(amount);

        }

    }

    public class StripeAdapter : IPaymentProcessor

    {

        private StripeGateway \_stripe = new StripeGateway();

        public void ProcessPayment(string amount)

        {

            \_stripe.SendPayment(amount);

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            IPaymentProcessor paypal = new PayPalAdapter();

            paypal.ProcessPayment("$100");

            IPaymentProcessor stripe = new StripeAdapter();

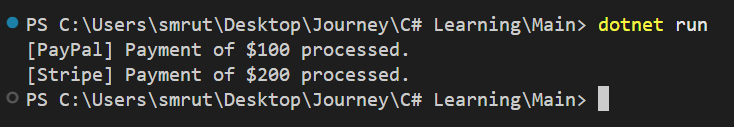
            stripe.ProcessPayment("$200");

        }

    }

}

**OUTPUT**

****

**Exercise 5: Implementing the Decorator Pattern**

**Solution**

**Filename – Exercise\_5.cs**

using System;

namespace DecoratorPatternExample

{

    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;

        protected NotifierDecorator(INotifier notifier)

        {

            \_wrappee = notifier;

        }

        public virtual void Send(string message)

        {

            \_wrappee.Send(message);

        }

    }

    public class SMSNotifierDecorator : NotifierDecorator

    {

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

        public override void Send(string message)

        {

            base.Send(message);

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

        }

    }

    public class SlackNotifierDecorator : NotifierDecorator

    {

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

        public override void Send(string message)

        {

            base.Send(message);

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

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            // Base notifier

            INotifier notifier = new EmailNotifier();

            // Add SMS and Slack via decorators

            notifier = new SMSNotifierDecorator(notifier);

            notifier = new SlackNotifierDecorator(notifier);

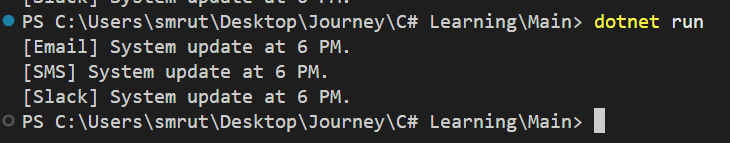
            notifier.Send("System update at 6 PM.");

        }

    }

}

**OUTPUT**

****

**Exercise 6: Implementing the Proxy Pattern**

**Solution**

**Filename – Exercise\_6.cs**

using System;

namespace ProxyPatternExample

{

    public interface IImage

    {

        void Display();

    }

    public class RealImage : IImage

    {

        private string \_filename;

        public RealImage(string filename)

        {

            \_filename = filename;

            LoadFromRemoteServer();

        }

        private void LoadFromRemoteServer()

        {

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

        }

        public void Display()

        {

            Console.WriteLine($"Displaying image '{\_filename}'");

        }

    }

    public class ProxyImage : IImage

    {

        private string \_filename;

        private RealImage \_realImage;

        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 image1 = new ProxyImage("nature1.jpg");

            IImage image2 = new ProxyImage("nature2.jpg");

            image1.Display();

            image1.Display();

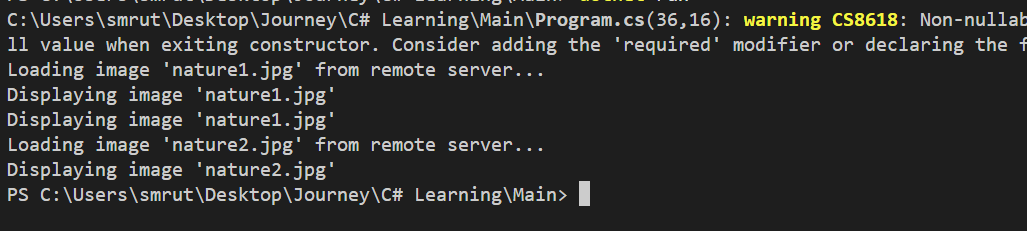
            image2.Display();

        }

    }

}

**OUTPUT**

****

**Exercise 7: Implementing the Observer Pattern**

**Solution**

**Filename – Exercise\_7.cs**

using System;

using System.Collections.Generic;

namespace ObserverPatternExample

{

    public interface IObserver

    {

        void Update(string stock, double price);

    }

    public interface IStock

    {

        void Register(IObserver observer);

        void Deregister(IObserver observer);

        void Notify(string stock, double price);

    }

    public class StockMarket : IStock

    {

        private List<IObserver> observers = new List<IObserver>();

        public void Register(IObserver observer)

        {

            observers.Add(observer);

        }

        public void Deregister(IObserver observer)

        {

            observers.Remove(observer);

        }

        public void Notify(string stock, double price)

        {

            foreach (var observer in observers)

            {

                observer.Update(stock, price);

            }

        }

        public void SetPrice(string stock, double price)

        {

            Console.WriteLine($"\nPrice updated: {stock} = {price}");

            Notify(stock, price);

        }

    }

    public class MobileApp : IObserver

    {

        public void Update(string stock, double price)

        {

            Console.WriteLine($"[MobileApp] {stock} updated to {price}");

        }

    }

    public class WebApp : IObserver

    {

        public void Update(string stock, double price)

        {

            Console.WriteLine($"[WebApp] {stock} updated to {price}");

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            StockMarket market = new StockMarket();

            IObserver mobileApp = new MobileApp();

            IObserver webApp = new WebApp();

            market.Register(mobileApp);

            market.Register(webApp);

            market.SetPrice("AAPL", 187.50);

            market.SetPrice("GOOGL", 2730.10);

            market.Deregister(webApp);

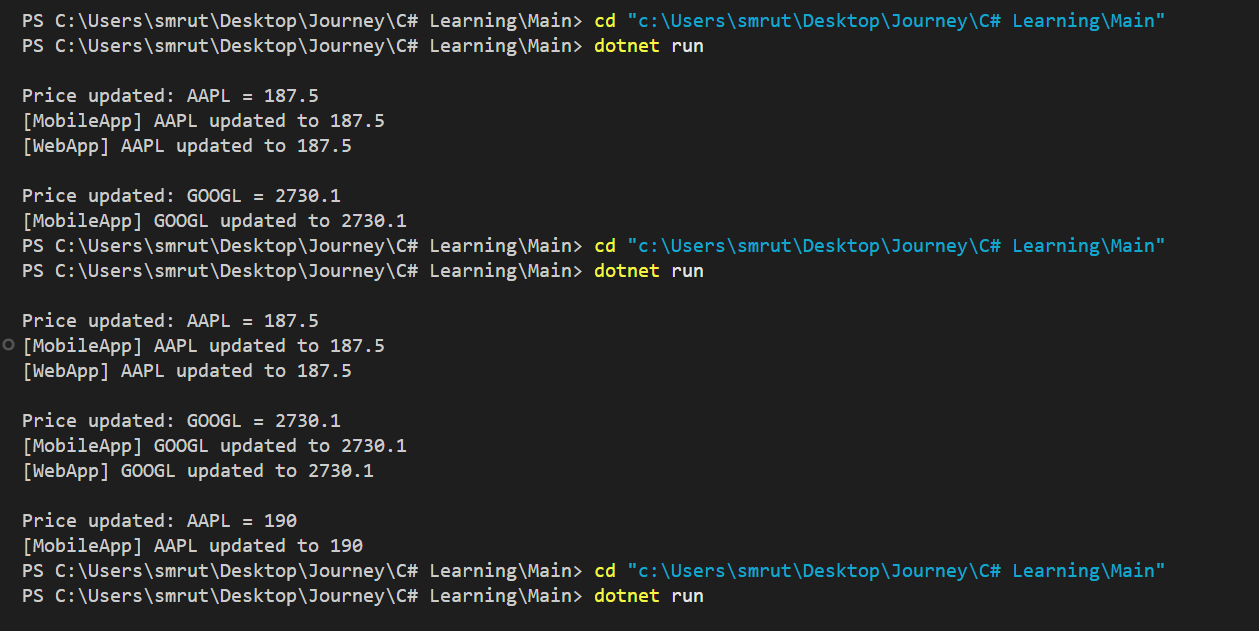
            market.SetPrice("AAPL", 190.00);

        }

    }

}

**OUTPUT**

****

**Exercise 8: Implementing the Stratergy Pattern**

**Solution**

**Filename – Exercise\_8.cs**

using System;

namespace StrategyPatternExample

{

    public interface IPaymentStrategy

    {

        void Pay(decimal amount);

    }

    public class CreditCardPayment : IPaymentStrategy

    {

        public void Pay(decimal amount)

        {

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

        }

    }

    public class PayPalPayment : IPaymentStrategy

    {

        public void Pay(decimal amount)

        {

            Console.WriteLine($"Paid {amount} using PayPal.");

        }

    }

    public class PaymentContext

    {

        private IPaymentStrategy \_strategy;

        public PaymentContext(IPaymentStrategy strategy)

        {

            \_strategy = strategy;

        }

        public void SetStrategy(IPaymentStrategy strategy)

        {

            \_strategy = strategy;

        }

        public void MakePayment(decimal amount)

        {

            \_strategy.Pay(amount);

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            PaymentContext context = new PaymentContext(new CreditCardPayment());

            context.MakePayment(500);

            context.SetStrategy(new PayPalPayment());

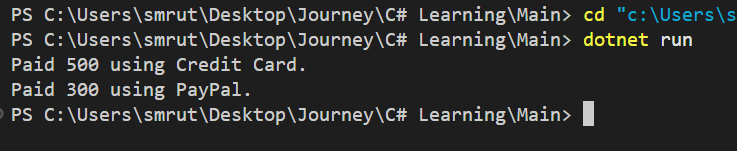
            context.MakePayment(300);

        }

    }

}

**OUTPUT**

****

**Exercise 9: Implementing the Command Pattern**

**Solution**

**Filename – Exercise\_9.cs**

using System;

namespace CommandPatternExample

{

    public interface ICommand

    {

        void Execute();

    }

    public class Light

    {

        public void TurnOn()

        {

            Console.WriteLine("The light is ON");

        }

        public void TurnOff()

        {

            Console.WriteLine("The 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();

        }

    }

    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);

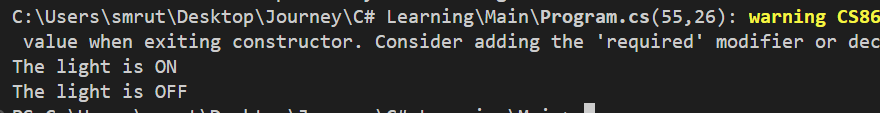
            remote.PressButton();

        }

    }

}

**OUTPUT**

****

**Exercise 10: Implementing the MVC Pattern**

**Solution**

**Filename – Exercise\_10.cs**

using System;

namespace MVCPatternExample

{

    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 Details:\nName: {name}\nID: {id}\nGrade: {grade}");

        }

    }

    public class StudentController

    {

        private Student \_student;

        private StudentView \_view;

        public StudentController(Student student, StudentView view)

        {

            \_student = student;

            \_view = view;

        }

        public void SetStudentName(string name) => \_student.Name = name;

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

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

        public void UpdateView()

        {

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

        }

    }

    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("Bob");

            controller.SetStudentGrade("B+");

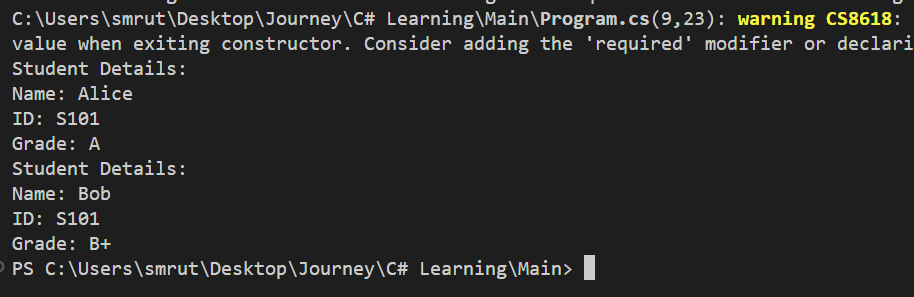
            controller.UpdateView();

        }

    }

}

**OUTPUT**

****

**Exercise 11: Implementing the Dependency Pattern**

**Solution**

**Filename – Exercise\_11.cs**

using System;

namespace DependencyInjectionExample

{

    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 GetCustomer(int id)

        {

            string customer = \_repository.FindCustomerById(id);

            Console.WriteLine(customer);

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            ICustomerRepository repo = new CustomerRepositoryImpl();

            CustomerService service = new CustomerService(repo);

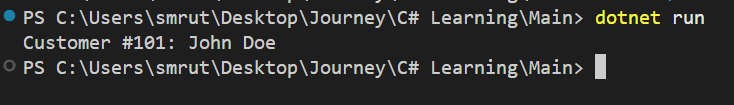
            service.GetCustomer(101);

        }

    }

}

**OUTPUT**

****