**Exercise 1: Implementing the Singleton Pattern**

**Code:**

using System;

using System.Threading.Tasks; // Add this line for Parallel class

namespace SingletonPatternExample

{

    // Singleton Logger class

    public sealed class Logger

    {

        // Private static instance

        private static Logger \_instance = null;

        // Lock object for thread safety

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

        // Private constructor

        private Logger()

        {

            Console.WriteLine("Logger instance created");

        }

        // Public static property to access the instance

        public static Logger Instance

        {

            get

            {

                // Double-check locking for thread safety

                if (\_instance == null)

                {

                    lock (\_lock)

                    {

                        if (\_instance == null)

                        {

                            \_instance = new Logger();

                        }

                    }

                }

                return \_instance;

            }

        }

        // Log method

        public void Log(string message)

        {

            Console.WriteLine($"[{DateTime.Now:yyyy-MM-dd HH:mm:ss}] {message}");

        }

    }

    // Test class

    class Program

    {

        static void Main(string[] args)

        {

            Console.WriteLine("Singleton Pattern Demo");

            Console.WriteLine("----------------------");

            // Get first instance

            Logger logger1 = Logger.Instance;

            logger1.Log("First log message");

            // Get second instance

            Logger logger2 = Logger.Instance;

            logger2.Log("Second log message");

            // Verify both references point to same instance

            Console.WriteLine($"\nAre logger1 and logger2 the same instance? {logger1 == logger2}");

            // Test thread safety

            Console.WriteLine("\nTesting thread safety...");

            Parallel.Invoke(

                () => {

                    Logger logger3 = Logger.Instance;

                    logger3.Log("Message from thread 1");

                },

                () => {

                    Logger logger4 = Logger.Instance;

                    logger4.Log("Message from thread 2");

                }

            );

            Console.WriteLine("\nPress any key to exit...");

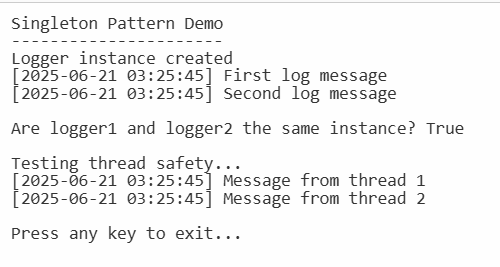
            Console.ReadKey();

        }

    }

}

**Output:**



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

**Codes:**

using System;

// Step 2: Define Document Classes

public interface IDocument

{

    void Open();

}

// Step 3: Create Concrete Document Classes

public class WordDocument : IDocument

{

    public void Open()

    {

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

    }

}

public class PdfDocument : IDocument

{

    public void Open()

    {

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

    }

}

public class ExcelDocument : IDocument

{

    public void Open()

    {

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

    }

}

// Step 4: Implement the Factory Method

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();

    }

}

// Step 5: Test the Factory Method Implementation

class Program

{

    static void Main(string[] args)

    {

        Console.WriteLine("Factory Method Pattern Demo");

        Console.WriteLine("---------------------------");

        // Create a Word document

        DocumentFactory wordFactory = new WordDocumentFactory();

        IDocument wordDoc = wordFactory.CreateDocument();

        wordDoc.Open();

        // Create a PDF document

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        IDocument pdfDoc = pdfFactory.CreateDocument();

        pdfDoc.Open();

        // Create an Excel document

        DocumentFactory excelFactory = new ExcelDocumentFactory();

        IDocument excelDoc = excelFactory.CreateDocument();

        excelDoc.Open();

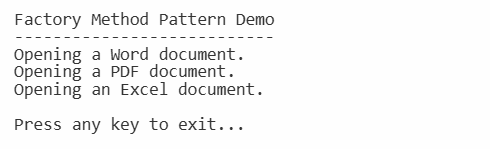
        Console.WriteLine("\nPress any key to exit...");

        Console.ReadKey();

    }

}

**Output:**



**Exercise 3: Implementing the Builder Pattern**

**Codes:**

using System;

// Step 2: Define a Product Class

public class Computer

{

    // Attributes of the Computer class

    public string CPU { get; private set; }

    public string RAM { get; private set; }

    public string Storage { get; private set; }

    public string GPU { get; private set; }

    public string OS { get; private set; }

    // Private constructor that takes the Builder as a parameter

    private Computer(Builder builder)

    {

        CPU = builder.CPU;

        RAM = builder.RAM;

        Storage = builder.Storage;

        GPU = builder.GPU;

        OS = builder.OS;

    }

    // Step 3: Implement the Builder Class

    public class Builder

    {

        // Optional parameters with default values

        public string CPU { get; private set; } = "Default CPU";

        public string RAM { get; private set; } = "8GB";

        public string Storage { get; private set; } = "256GB SSD";

        public string GPU { get; private set; } = "Integrated Graphics";

        public string OS { get; private set; } = "Windows 10";

        // Methods to set each attribute

        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 Builder SetOS(string os)

        {

            OS = os;

            return this;

        }

        // Build method that returns an instance of Computer

        public Computer Build()

        {

            return new Computer(this);

        }

    }

    // Override ToString for easy display of Computer details

    public override string ToString()

    {

        return $"Computer Specifications:\n" +

               $"CPU: {CPU}\n" +

               $"RAM: {RAM}\n" +

               $"Storage: {Storage}\n" +

               $"GPU: {GPU}\n" +

               $"OS: {OS}";

    }

}

// Step 5: Test the Builder Implementation

class Program

{

    static void Main(string[] args)

    {

        Console.WriteLine("Builder Pattern Demo");

        Console.WriteLine("---------------------");

        // Create a Computer using the Builder pattern

        Computer gamingPC = new Computer.Builder()

            .SetCPU("Intel i9")

            .SetRAM("32GB")

            .SetStorage("1TB SSD")

            .SetGPU("NVIDIA RTX 3080")

            .SetOS("Windows 11")

            .Build();

        Console.WriteLine(gamingPC);

        // Create another Computer with different configuration

        Computer officePC = new Computer.Builder()

            .SetCPU("AMD Ryzen 5")

            .SetRAM("16GB")

            .SetStorage("512GB SSD")

            .SetOS("Windows 10")

            .Build();

        Console.WriteLine("\n" + officePC);

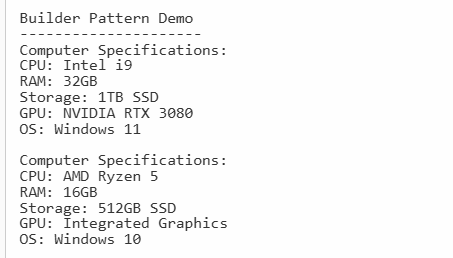
        Console.WriteLine("\nPress any key to exit...");

        Console.ReadKey();

    }

}

**Output:**



**Exercise 4: Implementing the Adapter Pattern**

**Codes:**

using System;

// Step 2: Define Target Interface

public interface IPaymentProcessor

{

    void ProcessPayment(decimal amount);

}

// Step 3: Implement Adaptee Classes (Different Payment Gateways with their own APIs)

public class PayPalGateway

{

    public void PayWithPayPal(decimal amount)

    {

        Console.WriteLine($"Processing payment of ${amount} through PayPal.");

    }

}

public class StripeGateway

{

    public void MakeStripePayment(decimal amountInCents)

    {

        Console.WriteLine($"Processing payment of ${amountInCents / 100m} through Stripe.");

    }

}

public class SquareGateway

{

    public void SquarePay(double amount)

    {

        Console.WriteLine($"Processing payment of ${amount} through Square.");

    }

}

// Step 4: Implement Adapter Classes

public class PayPalAdapter : IPaymentProcessor

{

    private readonly PayPalGateway \_payPalGateway;

    public PayPalAdapter(PayPalGateway payPalGateway)

    {

        \_payPalGateway = payPalGateway;

    }

    public void ProcessPayment(decimal amount)

    {

        \_payPalGateway.PayWithPayPal(amount);

    }

}

public class StripeAdapter : IPaymentProcessor

{

    private readonly StripeGateway \_stripeGateway;

    public StripeAdapter(StripeGateway stripeGateway)

    {

        \_stripeGateway = stripeGateway;

    }

    public void ProcessPayment(decimal amount)

    {

        // Stripe expects amount in cents as int

        int amountInCents = (int)(amount \* 100);

        \_stripeGateway.MakeStripePayment(amountInCents);

    }

}

public class SquareAdapter : IPaymentProcessor

{

    private readonly SquareGateway \_squareGateway;

    public SquareAdapter(SquareGateway squareGateway)

    {

        \_squareGateway = squareGateway;

    }

    public void ProcessPayment(decimal amount)

    {

        // Square accepts double

        \_squareGateway.SquarePay((double)amount);

    }

}

// Step 5: Test the Adapter Implementation

class Program

{

    static void Main(string[] args)

    {

        decimal paymentAmount = 100.50m;

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

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

        IPaymentProcessor squareProcessor = new SquareAdapter(new SquareGateway());

        Console.WriteLine("Using PayPal Adapter:");

        paypalProcessor.ProcessPayment(paymentAmount);

        Console.WriteLine("\nUsing Stripe Adapter:");

        stripeProcessor.ProcessPayment(paymentAmount);

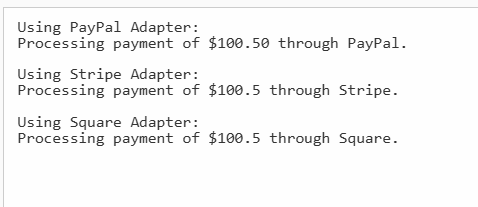
        Console.WriteLine("\nUsing Square Adapter:");

        squareProcessor.ProcessPayment(paymentAmount);

    }

}

**Output:**



**Exercise 5: Implementing the Decorator Pattern**

**Codes:**

using System;

// Step 2: Define Component Interface

public interface INotifier

{

    void Send(string message);

}

// Step 3: Implement Concrete Component

public class EmailNotifier : INotifier

{

    public void Send(string message)

    {

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

    }

}

// Step 4: Implement Decorator Classes

// Abstract Decorator Class

public abstract class NotifierDecorator : INotifier

{

    protected INotifier \_wrappee;

    public NotifierDecorator(INotifier notifier)

    {

        \_wrappee = notifier;

    }

    public virtual void Send(string message)

    {

        \_wrappee.Send(message);

    }

}

// Concrete Decorator - SMS

public class SMSNotifierDecorator : NotifierDecorator

{

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

    public override void Send(string message)

    {

        base.Send(message); // send notification via wrapped notifier

        SendSMS(message);   // add SMS notification

    }

    private void SendSMS(string message)

    {

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

    }

}

// Concrete Decorator - Slack

public class SlackNotifierDecorator : NotifierDecorator

{

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

    public override void Send(string message)

    {

        base.Send(message); // send notification via wrapped notifier

        SendSlack(message); // add Slack notification

    }

    private void SendSlack(string message)

    {

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

    }

}

// Step 5: Test the Decorator Implementation

class Program

{

    static void Main(string[] args)

    {

        string message = "This is a test notification.";

        // Base notifier - only Email

        INotifier emailNotifier = new EmailNotifier();

        Console.WriteLine("Email only:");

        emailNotifier.Send(message);

        Console.WriteLine("\nEmail + SMS:");

        // Email + SMS

        INotifier emailAndSmsNotifier = new SMSNotifierDecorator(emailNotifier);

        emailAndSmsNotifier.Send(message);

        Console.WriteLine("\nEmail + SMS + Slack:");

        // Email + SMS + Slack

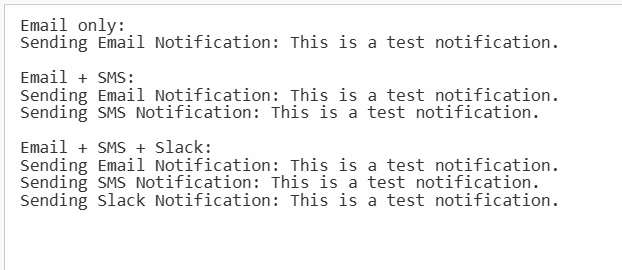
        INotifier emailSmsSlackNotifier = new SlackNotifierDecorator(emailAndSmsNotifier);

        emailSmsSlackNotifier.Send(message);

    }

}

**Output:**



**Exercise 6: Implementing the Proxy Pattern**

**Codes:**

using System;

// Step 2: Define Subject Interface

public interface IImage

{

    void Display();

}

// Step 3: Implement Real Subject Class

public class RealImage : IImage

{

    private string \_fileName;

    public RealImage(string fileName)

    {

        \_fileName = fileName;

        LoadFromRemoteServer(\_fileName);

    }

    private void LoadFromRemoteServer(string fileName)

    {

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

        // Simulate delay for loading

        System.Threading.Thread.Sleep(1000);

    }

    public void Display()

    {

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

    }

}

// Step 4: Implement Proxy Class

public class ProxyImage : IImage

{

    private RealImage \_realImage;

    private string \_fileName;

    public ProxyImage(string fileName)

    {

        \_fileName = fileName;

    }

    public void Display()

    {

        // Lazy initialization: only load when needed

        if (\_realImage == null)

        {

            \_realImage = new RealImage(\_fileName);

        }

        else

        {

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

        }

        \_realImage.Display();

    }

}

// Step 5: Test Proxy Implementation

class Program

{

    static void Main(string[] args)

    {

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

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

        // Image loads only on first display call

        image1.Display();

        Console.WriteLine();

        // Second call uses cached image - no loading delay

        image1.Display();

        Console.WriteLine();

        // Load second image

        image2.Display();

        Console.WriteLine();

        // Cached second image display

        image2.Display();

    }

}

**Output:**



**Exercise 7: Implementing the Observer Pattern**

**Codes:**

using System;

using System.Collections.Generic;

// Step 2: Define Subject Interface

public interface IStock

{

    void RegisterObserver(IObserver observer);

    void RemoveObserver(IObserver observer);

    void NotifyObservers();

}

// Step 4: Define Observer Interface

public interface IObserver

{

    void Update(decimal price);

}

// Step 3: Implement Concrete Subject

public class StockMarket : IStock

{

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

    private decimal \_stockPrice;

    public decimal StockPrice

    {

        get => \_stockPrice;

        set

        {

            \_stockPrice = value;

            NotifyObservers();

        }

    }

    public void RegisterObserver(IObserver observer)

    {

        if (!observers.Contains(observer))

            observers.Add(observer);

    }

    public void RemoveObserver(IObserver observer)

    {

        if (observers.Contains(observer))

            observers.Remove(observer);

    }

    public void NotifyObservers()

    {

        foreach (var observer in observers)

        {

            observer.Update(\_stockPrice);

        }

    }

}

// Step 5: Implement Concrete Observers

public class MobileApp : IObserver

{

    private string \_appName;

    public MobileApp(string appName)

    {

        \_appName = appName;

    }

    public void Update(decimal price)

    {

        Console.WriteLine($"MobileApp {\_appName} received stock price update: ${price}");

    }

}

public class WebApp : IObserver

{

    private string \_websiteName;

    public WebApp(string websiteName)

    {

        \_websiteName = websiteName;

    }

    public void Update(decimal price)

    {

        Console.WriteLine($"WebApp {\_websiteName} received stock price update: ${price}");

    }

}

// Step 6: Test the Observer Implementation

class Program

{

    static void Main(string[] args)

    {

        StockMarket stockMarket = new StockMarket();

        IObserver mobileApp1 = new MobileApp("MobileAppOne");

        IObserver mobileApp2 = new MobileApp("MobileAppTwo");

        IObserver webApp = new WebApp("FinanceWeb");

        stockMarket.RegisterObserver(mobileApp1);

        stockMarket.RegisterObserver(mobileApp2);

        stockMarket.RegisterObserver(webApp);

        Console.WriteLine("Updating stock price to 150.75");

        stockMarket.StockPrice = 150.75m;

        Console.WriteLine("\nRemoving MobileAppTwo observer");

        stockMarket.RemoveObserver(mobileApp2);

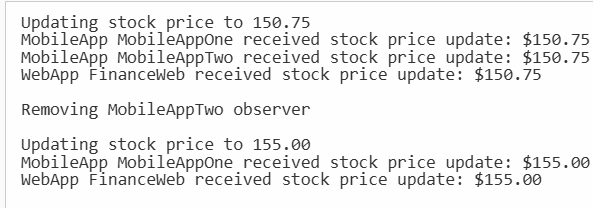
        Console.WriteLine("\nUpdating stock price to 155.00");

        stockMarket.StockPrice = 155.00m;

    }

}

**Output:**



**Exercise 8: Implementing the Strategy Pattern**

**Codes:**

using System;

namespace StrategyPatternExample

{

    // Step 1: Define Strategy Interface

    public interface IPaymentStrategy

    {

        void Pay(decimal amount);

    }

    // Step 2: Implement Concrete Strategies

    public class CreditCardPayment : IPaymentStrategy

    {

        private string \_cardNumber;

        public CreditCardPayment(string cardNumber)

        {

            \_cardNumber = cardNumber;

        }

        public void Pay(decimal amount)

        {

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

        }

    }

    public class PayPalPayment : IPaymentStrategy

    {

        private string \_email;

        public PayPalPayment(string email)

        {

            \_email = email;

        }

        public void Pay(decimal amount)

        {

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

        }

    }

    // Step 3: Implement Context Class

    public class PaymentContext

    {

        private IPaymentStrategy \_paymentStrategy;

        public void SetPaymentStrategy(IPaymentStrategy paymentStrategy)

        {

            \_paymentStrategy = paymentStrategy;

        }

        public void ExecutePayment(decimal amount)

        {

            if (\_paymentStrategy == null)

            {

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

                return;

            }

            \_paymentStrategy.Pay(amount);

        }

    }

    // Step 4: Test the Strategy Implementation

    class Program

    {

        static void Main(string[] args)

        {

            PaymentContext paymentContext = new PaymentContext();

            // Using Credit Card Payment

            paymentContext.SetPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456"));

            paymentContext.ExecutePayment(100.00m);

            // Using PayPal Payment

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

            paymentContext.ExecutePayment(200.00m);

            // Attempting to execute payment without setting a strategy

            paymentContext.SetPaymentStrategy(null);

            paymentContext.ExecutePayment(50.00m);

            Console.WriteLine("\nPress any key to exit...");

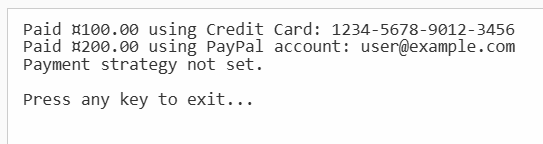
            Console.ReadKey();

        }

    }

}

**Output:**



**Exercise 9: Implementing the Command Pattern**

**Codes:**

using System;

// Step 2: Define Command Interface

public interface ICommand

{

    void Execute();

}

// Step 5: Implement Receiver Class

public class Light

{

    public void On()

    {

        Console.WriteLine("Light is ON");

    }

    public void Off()

    {

        Console.WriteLine("Light is OFF");

    }

}

// Step 3: Implement Concrete Commands

public class LightOnCommand : ICommand

{

    private Light \_light;

    public LightOnCommand(Light light)

    {

        \_light = light;

    }

    public void Execute()

    {

        \_light.On();

    }

}

public class LightOffCommand : ICommand

{

    private Light \_light;

    public LightOffCommand(Light light)

    {

        \_light = light;

    }

    public void Execute()

    {

        \_light.Off();

    }

}

// Step 4: Implement Invoker Class

public class RemoteControl

{

    private ICommand \_command;

    public void SetCommand(ICommand command)

    {

        \_command = command;

    }

    public void PressButton()

    {

        if (\_command != null)

        {

            \_command.Execute();

        }

        else

        {

            Console.WriteLine("No command assigned.");

        }

    }

}

// Step 6: Test the Command Implementation

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();

        // Turn light ON

        remote.SetCommand(lightOn);

        remote.PressButton();

        // Turn light OFF

        remote.SetCommand(lightOff);

        remote.PressButton();

    }

}

**Output:**



**Exercise 10: Implementing the MVC Pattern**

**Codes:**

using System;

// Step 2: Model Class

public class Student

{

    public string Name { get; set; }

    public int Id { get; set; }

    public string Grade { get; set; }

}

// Step 3: View Class

public class StudentView

{

    public void DisplayStudentDetails(Student student)

    {

        Console.WriteLine("Student Details:");

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

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

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

    }

}

// Step 4: Controller Class

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()

    {

        return \_model.Name;

    }

    public void SetStudentId(int id)

    {

        \_model.Id = id;

    }

    public int GetStudentId()

    {

        return \_model.Id;

    }

    public void SetStudentGrade(string grade)

    {

        \_model.Grade = grade;

    }

    public string GetStudentGrade()

    {

        return \_model.Grade;

    }

    public void UpdateView()

    {

        \_view.DisplayStudentDetails(\_model);

    }

}

// Step 5: Test MVC Implementation

class Program

{

    static void Main(string[] args)

    {

        // Create Model

        Student student = new Student { Name = "Alice", Id = 101, Grade = "A" };

        // Create View

        StudentView view = new StudentView();

        // Create Controller

        StudentController controller = new StudentController(student, view);

        // Display initial details

        controller.UpdateView();

        Console.WriteLine("\nUpdating student details...\n");

        // Update student details via controller

        controller.SetStudentName("Bob");

        controller.SetStudentGrade("B+");

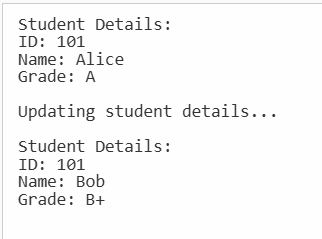
        // Display updated details

        controller.UpdateView();

    }

}

**Output:**



**Exercise 11: Implementing Dependency Injection**

**Codes:**

using System;

namespace DependencyInjectionExample

{

    // Step 1: Define Repository Interface

    public interface ICustomerRepository

    {

        Customer FindCustomerById(int id);

    }

    // Step 2: Implement Concrete Repository

    public class CustomerRepositoryImpl : ICustomerRepository

    {

        public Customer FindCustomerById(int id)

        {

            // Simulating a customer lookup

            if (id == 1)

            {

                return new Customer { Id = 1, Name = "John Doe", Email = "john.doe@example.com" };

            }

            else

            {

                return null; // Customer not found

            }

        }

    }

    // Model Class

    public class Customer

    {

        public int Id { get; set; }

        public string Name { get; set; }

        public string Email { get; set; }

    }

    // Step 3: Define Service Class

    public class CustomerService

    {

        private readonly ICustomerRepository \_customerRepository;

        // Step 4: Implement Dependency Injection via Constructor

        public CustomerService(ICustomerRepository customerRepository)

        {

            \_customerRepository = customerRepository;

        }

        public Customer GetCustomerById(int id)

        {

            return \_customerRepository.FindCustomerById(id);

        }

    }

    // Step 5: Test the Dependency Injection Implementation

    class Program

    {

        static void Main(string[] args)

        {

            // Create an instance of the repository

            ICustomerRepository customerRepository = new CustomerRepositoryImpl();

            // Inject the repository into the service

            CustomerService customerService = new CustomerService(customerRepository);

            // Find a customer by ID

            Customer customer = customerService.GetCustomerById(1);

            // Display customer details

            if (customer != null)

            {

                Console.WriteLine($"Customer ID: {customer.Id}");

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

                Console.WriteLine($"Customer Email: {customer.Email}");

            }

            else

            {

                Console.WriteLine("Customer not found.");

            }

            Console.WriteLine("\nPress any key to exit...");

            Console.ReadKey();

        }

    }

}

**Output:**

