**Exercise 1: Implementing the Singleton Pattern**

**Implementation**

public sealed class Logger

{

    private static Logger instance = null;

    private static readonly object padlock = new object();

    // Private constructor ensures it can't be instantiated outside this class

    private Logger()

    {

        Console.WriteLine("Logger Initialized");

    }

    // Public static method to get the instance

    public static Logger Instance

    {

        get

        {

            lock (padlock) // Thread safety

            {

                if (instance == null)

                {

                    instance = new Logger();

                }

                return instance;

            }

        }

    }

    public void Log(string message)

    {

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

    }

}

class Program

{

    static void Main(string[] args)

    {

        Logger logger1 = Logger.Instance;

        logger1.Log("First message");

        Logger logger2 = Logger.Instance;

        logger2.Log("Second message");

        // Verify both references point to the same instance

        if (ReferenceEquals(logger1, logger2))

        {

            Console.WriteLine("Only one instance of Logger exists.");

        }

        else

        {

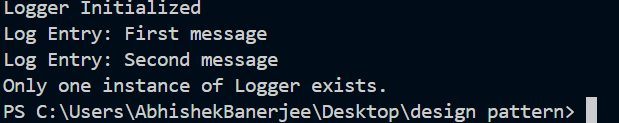
            Console.WriteLine("Multiple instances detected!");

        }

    }

}

**OUTPUT**



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

**Implementation**

//defining interface document

public interface Document

{

    void type();

}

//defining concrete classes for different document types

public class WordDocument : Document

{

    public void type()

    {

        Console.WriteLine("This is a Word document.");

    }

}

public class PdfDocument : Document

{

    public void type()

    {

        Console.WriteLine("This is a PDF document.");

    }

}

public class ExcelDocument: Document

{

    public void type()

    {

        Console.WriteLine("This is an Excel document.");

    }

}

//defining factory class to create document instances

public class DocumentFactory

{

       public static Document CreateDocument(string type)

    {

        switch (type.ToLower())

        {

            case "word":

                return new WordDocument();

            case "pdf":

                return new PdfDocument();

            case "excel":

                return new ExcelDocument();

            default:

                return new WordDocument(); // Default to WordDocument if type is unknown

        }

    }

}

// Main program to demonstrate the factory pattern

public class Program

{

    public static void Main(string[] args)

    {

        Document doc1 = DocumentFactory.CreateDocument("word");

        doc1.type(); // Output: This is a Word document.

        Document doc2 = DocumentFactory.CreateDocument("pdf");

        doc2.type(); // Output: This is a PDF document.

        Document doc3 = DocumentFactory.CreateDocument("excel");

        doc3.type(); // Output: This is an Excel document.

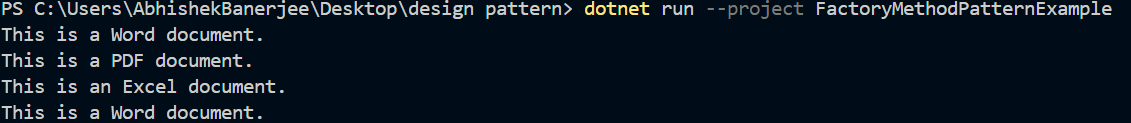
        Document doc4 = DocumentFactory.CreateDocument("unknown");

        doc4.type(); // Output: This is a Word document.

    }

}

**OUTPUT**



**Exercise 3: Implementing the Builder Pattern**

**Implementation**

public class BuildProduct

{

    public string CPU;

    public string RAM;

    public string Storage;

    public BuildProduct setCPU(string cpu)

    {

        this.CPU = cpu;

        return this;

    }

    public BuildProduct setRAM(string ram)

    {

        this.RAM = ram;

        return this;

    }

    public BuildProduct setStorage(string storage)

    {

        this.Storage = storage;

        return this;

    }

    public Product build()

    {

        return new Product(this);

    }

}

public class Product

{

    public string CPU;

    public string RAM;

    public string Storage;

    public Product(BuildProduct product)

    {

        this.CPU = product.CPU;

        this.RAM = product.RAM;

        this.Storage = product.Storage;

    }

}

public class Program

{

    public static void Main(string[] args)

    {

        BuildProduct build = new BuildProduct();

        Product product = build.setCPU("Intel i7")

                               .setRAM("16GB")

                               .setStorage("512GB SSD")

                               .build();

        Console.WriteLine("Product built with:");

        Console.WriteLine($"CPU: {product.CPU}");

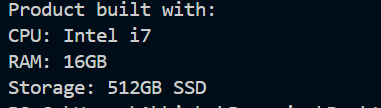
        Console.WriteLine($"RAM: {product.RAM}");

        Console.WriteLine($"Storage: {product.Storage}");

    }

}

**OUTPUT**

****

**Exercise 4: Implementing the Adapter Pattern**

**Implementation**

// target payement interface

public interface IPaymentProcessor

{

    void ProcessPayment(decimal amount);

}

//define adaptee class for PayPalGateway

public class PayPalGateway

{

    public void MakePayment(decimal amount)

    {

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

    }

}

//define adaptee class for StripeGateway

public class StripeGateway

{

    public void Charge(decimal amount)

    {

        Console.WriteLine($"Charged ₹{amount} using Stripe.");

    }

}

//Define adapter class for paypal

public class PayPalAdapter : IPaymentProcessor

{

    private readonly PayPalGateway \_paypal = new PayPalGateway();

    public void ProcessPayment(decimal amount)

    {

        \_paypal.MakePayment(amount);

    }

}

//define adapter class for Stripe

public class StripeAdapter : IPaymentProcessor

{

    private readonly StripeGateway \_stripe = new StripeGateway();

    public void ProcessPayment(decimal amount)

    {

        \_stripe.Charge(amount);

    }

}

//Main program to test the adapter implementation

public class Program

{

    public static void Main(string[] args)

    {

        IPaymentProcessor paypalProcessor = new PayPalAdapter();

        IPaymentProcessor stripeProcessor = new StripeAdapter();

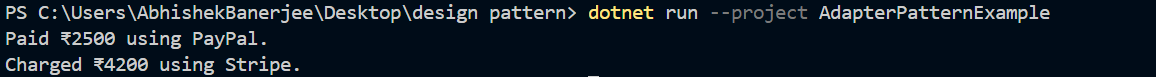
        paypalProcessor.ProcessPayment(2500);

        stripeProcessor.ProcessPayment(4200);

    }

}

**OUTPUT**

****