**Design principles and Patterns**

**Exercise 1: Implementing the Singleton Pattern**

using System;

public class Singleton

{

private static Singleton instance = null;

private static readonly object padlock = new object();

// Private constructor

private Singleton()

{

}

public static Singleton GetInstance()

{

// Thread-safe lazy initialization

if (instance == null)

{

lock (padlock)

{

if (instance == null)

{

instance = new Singleton();

}

}

}

return instance;

}

public void ShowMessage()

{

Console.WriteLine("Hello from Singleton!");

}

}

class Program

{

static void Main(string[] args)

{

Singleton singleton = Singleton.GetInstance();

singleton.ShowMessage();

}

}

**Output:**



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

using System;

namespace FactoryPatternDemo

{

public interface IVehicle

{

void Operate();

}

public class Car : IVehicle

{

public void Operate()

{

Console.WriteLine("Driving a car.");

}

}

public class Motorcycle : IVehicle

{

public void Operate()

{

Console.WriteLine("Riding a motorcycle.");

}

}

public abstract class VehicleFactory

{

public abstract IVehicle CreateVehicle();

public void PerformOperation()

{

var vehicle = CreateVehicle();

vehicle.Operate();

}

}

public class CarFactory : VehicleFactory

{

public override IVehicle CreateVehicle()

{

return new Car();

}

}

public class MotorcycleFactory : VehicleFactory

{

public override IVehicle CreateVehicle()

{

return new Motorcycle();

}

}

public class Program

{

private static VehicleFactory \_factory;

public static void Main(string[] args)

{

Console.WriteLine("--- Configuring application for Cars ---");

Configure("Car");

RunBusinessLogic();

Console.WriteLine("\n--- Re-configuring application for Motorcycles ---");

Configure("Motorcycle");

RunBusinessLogic();

}

public static void Configure(string vehicleType)

{

switch (vehicleType)

{

case "Car":

\_factory = new CarFactory();

break; // The 'break' is required in a switch statement

case "Motorcycle":

\_factory = new MotorcycleFactory();

break;

default:

throw new ArgumentException("Unknown vehicle type");

}

}

public static void RunBusinessLogic()

{

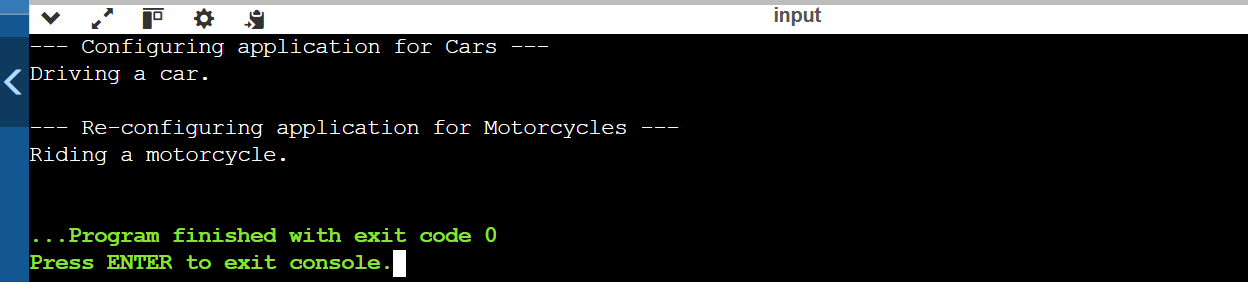
\_factory.PerformOperation();

}

}

}

**Output:**



**Data Structures and Algorithms**

**Exercise 2: E-commerce Platform Search Function**

using System;

using System.Collections.Generic;

using System.Linq;

public class Product

{

public string Name { get; set; }

public decimal Price { get; set; }

public Product(string name, decimal price)

{

Name = name;

Price = price;

}

}

public class ECommercePlatform

{

private List<Product> products;

public ECommercePlatform()

{

products = new List<Product>

{

new Product("Laptop", 999.99m),

new Product("Smartphone", 499.99m),

new Product("Tablet", 299.99m),

new Product("Headphones", 149.99m)

};

}

public List<Product> SearchProducts(string query)

{

if (string.IsNullOrEmpty(query))

{

return new List<Product>();

}

return products.Where(p => p.Name.IndexOf(query, StringComparison.OrdinalIgnoreCase) >= 0).ToList();

}

}

class Program

{

static void Main()

{

ECommercePlatform platform = new ECommercePlatform();

Console.WriteLine("Search for a product:");

string searchQuery = Console.ReadLine();

List<Product> results = platform.SearchProducts(searchQuery);

if (results.Count == 0)

{

Console.WriteLine("No products found.");

}

else

{

Console.WriteLine("Search Results:");

foreach (var product in results)

{

Console.WriteLine($"{product.Name} - ${product.Price}");

}

}

}

}

**Output:**

****

**Exercise 7: Financial Forecasting**

using System;

using System.Collections.Generic;

public class FinancialForecasting

{

private List<decimal> historicalPrices;

public FinancialForecasting(List<decimal> prices)

{

historicalPrices = prices;

}

public decimal PredictNextPrice(int period)

{

if (period <= 0 || period > historicalPrices.Count)

{

throw new ArgumentException("Invalid period. Period must be greater than 0 and less than or equal to the number of historical prices.");

}

int start = historicalPrices.Count - period;

decimal sum = 0;

for (int i = start; i < historicalPrices.Count; i++)

{

sum += historicalPrices[i];

}

return sum / period;

}

}

class Program

{

static void Main()

{

// Example historical prices

List<decimal> historicalPrices = new List<decimal>

{

100.00m, 102.50m, 105.00m, 107.50m, 110.00m,

112.50m, 115.00m, 117.50m, 120.00m, 122.50m

};

FinancialForecasting forecasting = new FinancialForecasting(historicalPrices);

Console.WriteLine("Enter the period for the moving average:");

int period = int.Parse(Console.ReadLine());

try

{

decimal predictedPrice = forecasting.PredictNextPrice(period);

Console.WriteLine($"Predicted next day's price: ${predictedPrice}");

}

catch (ArgumentException ex)

{

Console.WriteLine(ex.Message);

}

}

}

