WEEK 1_(Superset Id : 6363523)

EXERCISE - 1 (Implementing the Singleton Pattern)

```
using System;

public sealed class Singleton
{
    private static readonly Lazy<Singleton> _instance = new Lazy<Singleton>(() => new
Singleton());

    private Singleton()
    {
        Console.WriteLine("Singleton instance created");
    }

    public static Singleton Instance
    {
        get
        {
            return _instance.Value;
        }
    }

    public void DoSomething()
    {
        Console.WriteLine("Doing something with the singleton instance");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Singleton s1 = Singleton.Instance;
        s1.DoSomething();

        Singleton s2 = Singleton.Instance;
        Console.WriteLine(s1 == s2); // Outputs: True
    }
}
```

OUTPUT

```
Singleton instance created
Doing something with the singleton instance
```

True

EXERCISE 2(Implementing the Factory Method Pattern)

```
public interface IVehicle
{
    void Drive();
}
public class Car : IVehicle
{
    public void Drive()
    {
        Console.WriteLine("Driving a car...");
    }
}

public class Motorcycle : IVehicle
{
    public void Drive()
    {
        Console.WriteLine("Driving a motorcycle...");
    }
}

public class VehicleFactory
{
    public IVehicle CreateVehicle(string type)
    {
        if (type.ToLower() == "car")
        {
            return new Car();
        }
        else if (type.ToLower() == "motorcycle")
        {
            return new Motorcycle();
        }
        else
        {
            throw new ArgumentException("Unknown vehicle type.");
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        var factory = new VehicleFactory();
```

```
        IVehicle car = factory.CreateVehicle("car");
        car.Drive();  // Output: Driving a car...

        IVehicle bike = factory.CreateVehicle("motorcycle");
        bike.Drive(); // Output: Driving a motorcycle...
    }
}
```

OUTPUT

```
Driving a car...
Driving a motorcycle...
```

EXERCISE 3(E-commerce Platform Search Function)

```
public class Product
{
    public string Name { get; set; }
    public string Category { get; set; }
    public double Price { get; set; }
    public double Rating { get; set; }
}
using System.Collections.Generic;

public interface ISearchStrategy
{
    List<Product> Search(List<Product> products, string query);
}

public class NameSearch : ISearchStrategy
{
    public List<Product> Search(List<Product> products, string query)
    {
        return products.FindAll(p => p.Name.Contains(query,
StringComparison.OrdinalIgnoreCase));
    }
}
public class CategorySearch : ISearchStrategy
{
    public List<Product> Search(List<Product> products, string query)
    {
        return products.FindAll(p => p.Category.Equals(query,
StringComparison.OrdinalIgnoreCase));
    }
}
public class PriceSearch : ISearchStrategy
{
    public List<Product> Search(List<Product> products, string query)
```

```
    {
        var parts = query.Split('-');
        double min = double.Parse(parts[0]);
        double max = double.Parse(parts[1]);

        return products.FindAll(p => p.Price >= min && p.Price <= max);
    }
}
public class SearchFactory
{
    public static ISearchStrategy GetStrategy(string type)
    {
        return type.ToLower() switch
        {
            "name" => new NameSearch(),
            "category" => new CategorySearch(),
            "price" => new PriceSearch(),
            _ => throw new ArgumentException("Invalid search type")
        };
    }
}
public class SearchService
{
    public List<Product> PerformSearch(List<Product> products, string type, string query)
    {
        ISearchStrategy strategy = SearchFactory.GetStrategy(type);
        return strategy.Search(products, query);
    }
}
class Program
{
    static void Main()
    {
        List<Product> catalog = new List<Product>
        {
            new Product { Name = "iPhone", Category = "Electronics", Price = 999, Rating = 4.7
},
            new Product { Name = "Desk Chair", Category = "Furniture", Price = 150, Rating =
4.3 },
            new Product { Name = "Samsung TV", Category = "Electronics", Price = 799, Rating
= 4.6 }
        };

        var service = new SearchService();

        var results = service.PerformSearch(catalog, "category", "Electronics");
        foreach (var product in results)
        {
```

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

OUTPUT
iPhone - Electronics - $999
Samsung TV - Electronics - $799

EXERCISE 4(Financial Forecasting)

```
public class HistoricalData
{
    public List<double> MonthlyRevenue { get; set; }

    public HistoricalData(List<double> revenue)
    {
        MonthlyRevenue = revenue;
    }
}
public interface IForecastingStrategy
{
    List<double> Forecast(HistoricalData data, int monthsAhead);
}
public class LinearGrowthForecast : IForecastingStrategy
{
    public List<double> Forecast(HistoricalData data, int monthsAhead)
    {
        var results = new List<double>();
        var history = data.MonthlyRevenue;
        double last = history[^1];
        double growth = (history[^1] - history[0]) / (history.Count - 1);

        for (int i = 1; i <= monthsAhead; i++)
        {
            results.Add(last + (growth * i));
        }

        return results;
    }
}
public class MovingAverageForecast : IForecastingStrategy
{
    public List<double> Forecast(HistoricalData data, int monthsAhead)
    {
        var results = new List<double>();
```

```
        var history = data.MonthlyRevenue;
        int window = Math.Min(3, history.Count);

        double average = history.Skip(history.Count - window).Average();

        for (int i = 0; i < monthsAhead; i++)
        {
            results.Add(average);
        }

        return results;
    }
}
public static class ForecastFactory
{
    public static IForecastingStrategy GetStrategy(string type)
    {
        return type.ToLower() switch
        {
            "linear" => new LinearGrowthForecast(),
            "movingaverage" => new MovingAverageForecast(),
            _ => throw new ArgumentException("Unknown forecast type")
        };
    }
}
public class ForecastService
{
    public List<double> GenerateForecast(HistoricalData data, string method, int months)
    {
        var strategy = ForecastFactory.GetStrategy(method);
        return strategy.Forecast(data, months);
    }
}
class Program
{
    static void Main()
    {
        var pastRevenue = new List<double> { 10000, 12000, 14000, 16000 };
        var data = new HistoricalData(pastRevenue);

        var service = new ForecastService();
        var forecast = service.GenerateForecast(data, "linear", 3);

        Console.WriteLine("Forecasted Revenue:");
        forecast.ForEach(Console.WriteLine);
    }
}
```

OUTPUT

Forecasted Revenue:
18000
20000
22000