Lab Exercise 1: Implement a Simple Singleton

Task:

Implement a basic Singleton class in C#. Ensure that only one instance of the class can be created.

Solution:

```csharp
public class Singleton
{
    private static Singleton _instance;

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            if (_instance == null)
            {
                _instance = new Singleton();
            }
            return _instance;
        }
    }

    public void ShowMessage()
    {
        Console.WriteLine("Singleton instance created!");
    }
}

// Usage
```

class Program
{
    static void Main(string[] args)
    {
        Singleton instance = Singleton.Instance;
        instance.ShowMessage();
    }
}
~~~

Lab Exercise 2: Thread-Safe Singleton

Task:

Modify the Singleton class from Lab Exercise 1 to make it thread-safe.

Solution:

```csharp
public class Singleton
{
    private static Singleton _instance;
    private static readonly object _lock = new object();

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            lock (_lock)
            {
                if (_instance == null)
                {
~~~

```
                    _instance = new Singleton();
                }
                return _instance;
            }
        }
    }

    public void ShowMessage()
    {
        Console.WriteLine("Thread-safe Singleton instance created!");
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        Singleton instance = Singleton.Instance;
        instance.ShowMessage();
    }
}
```

Lab Exercise 3: Lazy Initialization Singleton

Task:

Implement a Singleton using the `Lazy<T>` type in C#.

Solution:

```csharp
public class Singleton
```

```
{
    private static readonly Lazy<Singleton> _instance = new Lazy<Singleton>(() => new Singleton());

    private Singleton() { }

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

    public void ShowMessage()
    {
        Console.WriteLine("Lazy Singleton instance created!");
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        Singleton instance = Singleton.Instance;
        instance.ShowMessage();
    }
}
```

Lab Exercise 4: Implement a Simple Factory Method

Task:

Create a Factory Method pattern to instantiate different types of products (e.g., `ProductA` and `ProductB`).

Solution:

```csharp
public interface IProduct
{
    string Operation();
}

public class ProductA : IProduct
{
    public string Operation()
    {
        return "Result of ProductA";
    }
}

public class ProductB : IProduct
{
    public string Operation()
    {
        return "Result of ProductB";
    }
}

public abstract class Creator
{
    public abstract IProduct FactoryMethod();
    public string SomeOperation()
    {
        var product = FactoryMethod();
        return "Creator: Working with " + product.Operation();
    }
}

public class ConcreteCreatorA : Creator
{
    public override IProduct FactoryMethod()
    {
        return new ProductA();
    }
}

public class ConcreteCreatorB : Creator
{
    public override IProduct FactoryMethod()
    {
        return new ProductB();
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        Creator creator = new ConcreteCreatorA();
        Console.WriteLine(creator.SomeOperation());
        creator = new ConcreteCreatorB();
        Console.WriteLine(creator.SomeOperation());
    }
}
```

Lab Exercise 5: Implement an Abstract Factory

Task:

Create an Abstract Factory pattern for creating related objects such as `Button` and `Checkbox` in different themes (e.g., `DarkTheme`, `LightTheme`).

Solution:

```csharp
public interface IButton
{
    void Paint();
}

public interface ICheckbox
{
    void Paint();
}

public class DarkButton : IButton
{
    public void Paint()
    {
        Console.WriteLine("Dark Button");
    }
}
```

```
public class LightButton : IButton
{
    public void Paint()
    {
        Console.WriteLine("Light Button");
    }
}

public class DarkCheckbox : ICheckbox
{
    public void Paint()
    {
        Console.WriteLine("Dark Checkbox");
    }
}

public class LightCheckbox : ICheckbox
{
    public void Paint()
    {
        Console.WriteLine("Light Checkbox");
    }
}

public interface IGUIFactory
{
    IButton CreateButton();
    ICheckbox CreateCheckbox();
}

public class DarkThemeFactory : IGUIFactory
```

```
{
    public IButton CreateButton()
    {
        return new DarkButton();
    }

    public ICheckbox CreateCheckbox()
    {
        return new DarkCheckbox();
    }
}

public class LightThemeFactory : IGUIFactory
{
    public IButton CreateButton()
    {
        return new LightButton();
    }

    public ICheckbox CreateCheckbox()
    {
        return new LightCheckbox();
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        IGUIFactory factory = new DarkThemeFactory();
        var button = factory.CreateButton();
        var checkbox = factory.CreateCheckbox();

        button.Paint();
        checkbox.Paint();

        factory = new LightThemeFactory();

        button = factory.CreateButton();
        checkbox = factory.CreateCheckbox();

        button.Paint();
        checkbox.Paint();
    }
}
```

Lab Exercise 6: Implement a Simple Builder

Task:

Implement a Builder pattern for constructing a `Pizza` object step by step.

Solution:

```csharp
public class Pizza
{
    public string Dough { get; set; }
    public string Sauce { get; set; }
    public string Topping { get; set; }

    public void ShowPizza()
    {
        Console.WriteLine($"Pizza with {Dough} dough, {Sauce} sauce, and {Topping} topping.");
    }
}

public interface IPizzaBuilder
{
    void BuildDough();
    void BuildSauce();
    void BuildTopping();
    Pizza GetPizza();
}

public class MargheritaPizzaBuilder : IPizzaBuilder
{
    private Pizza _pizza = new Pizza();

    public void BuildDough()
    {
        _pizza.Dough = "Soft";
    }

    public void BuildSauce()
    {
        _pizza.Sauce = "Tomato";
    }

    public void BuildTopping()
    {
        _pizza.Topping = "Cheese";
    }
    public Pizza GetPizza()
    {
        return _pizza;
    }
}

public class SpicyPizzaBuilder : IPizzaBuilder
{
    private Pizza _pizza = new Pizza();

    public void BuildDough()
    {
        _pizza.Dough = "Crispy";
    }

    public void BuildSauce()
    {
        _pizza.Sauce = "Hot";
    }

    public void BuildTopping()
    {
        _pizza.Topping = "Pepperoni";
    }

    public Pizza GetPizza()
    {
        return _pizza;
    }
}
```

```
public class Director
{
    private IPizzaBuilder _builder;

    public void SetBuilder(IPizzaBuilder builder)
    {
        _builder = builder;
    }

    public void BuildPizza()
    {
        _builder.BuildDough();
        _builder.BuildSauce();
        _builder.BuildTopping();
    }

    public Pizza GetPizza()
    {
        return _builder.GetPizza();
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        Director director = new Director();

        IPizzaBuilder margheritaBuilder = new MargheritaPizzaBuilder();
        director.SetBuilder(margheritaBuilder);
        director.BuildPizza();

        Pizza pizza = director.GetPizza();
        pizza.ShowPizza();

        IPizzaBuilder spicyBuilder = new SpicyPizzaBuilder();
        director.SetBuilder(spicyBuilder);
        director.BuildPizza();

        pizza = director.GetPizza();
        pizza.ShowPizza();
    }
}
```

Lab Exercise 7: Implement a Prototype Pattern

Task:

Create a Prototype pattern where you can clone a `Shape` object (e.g., `Circle`, `Rectangle`).

Solution:

```csharp
public abstract class Shape
{
    public abstract Shape Clone();
}

public class Circle : Shape
{
    public int Radius { get; set; }
    public Circle(int radius)
    {
        Radius = radius;
    }

    public override Shape Clone()
    {
        return (Shape)this.MemberwiseClone();
    }

    public void ShowShape()
    {
        Console.WriteLine($"Circle with radius {Radius}");
    }
}

public class Rectangle : Shape
{
    public int Width { get; set; }
    public int Height { get; set; }

    public Rectangle(int width, int height)
    {
        Width = width;
        Height = height;
    }

    public override Shape Clone()
    {
        return (Shape)this.MemberwiseClone();
    }
    public void ShowShape()
    {
        Console.WriteLine($"Rectangle with width {Width} and height {Height}");
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        Circle circle1 = new Circle(10);
        Circle circle2 = (Circle)circle1.Clone();
        circle2.Radius = 20;

        circle1.ShowShape();
        circle2.ShowShape();

        Rectangle rect1 = new Rectangle(5, 10);
        Rectangle rect2 = (Rectangle)rect1.Clone();
        rect2.Width = 15;

        rect1.ShowShape();
        rect2.ShowShape();
    }
}
```

Lab Exercise 8: Singleton with Initialization Parameters

Task:

}

Create a Singleton class that allows initialization with parameters (e.g., `Configuration

` with `Name` and `Version`).

Solution:

```csharp
public class Configuration
{
    private static Configuration _instance;
    private static readonly object _lock = new object();

    public string Name { get; private set; }
    public string Version { get; private set; }

    private Configuration(string name, string version)
    {
        Name = name;
        Version = version;
    }

    public static Configuration Instance(string name = null, string version = null)
    {
        lock (_lock)
        {
            if (_instance == null)
            {
                _instance = new Configuration(name, version);
            }
            return _instance;
        }
    }
    public void ShowConfig()
    {
        Console.WriteLine($"Configuration: {Name}, Version: {Version}");
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        Configuration config = Configuration.Instance("MyApp", "1.0");
        config.ShowConfig();

        Configuration anotherConfig = Configuration.Instance();
        anotherConfig.ShowConfig(); // Will show the same values
    }
}
```

Lab Exercise 9: Factory Method for Different Notification Types

Task:

Implement a Factory Method pattern to create different types of notifications (e.g., `EmailNotification`, `SMSNotification`).

Solution:

```csharp
public interface INotification
{
    void Notify(string message);
```

```
}

public class EmailNotification : INotification
{
    public void Notify(string message)
    {
        Console.WriteLine($"Sending Email: {message}");
    }
}

public class SMSNotification : INotification
{
    public void Notify(string message)
    {
        Console.WriteLine($"Sending SMS: {message}");
    }
}

public abstract class NotificationCreator
{
    public abstract INotification FactoryMethod();

    public void SendNotification(string message)
    {
        var notification = FactoryMethod();
        notification.Notify(message);
    }
}

public class EmailNotificationCreator : NotificationCreator
{
    public override INotification FactoryMethod()
    {
        return new EmailNotification();
    }
}

public class SMSNotificationCreator : NotificationCreator
{
    public override INotification FactoryMethod()
    {
        return new SMSNotification();
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        NotificationCreator creator = new EmailNotificationCreator();
        creator.SendNotification("Hello via Email!");

        creator = new SMSNotificationCreator();
        creator.SendNotification("Hello via SMS!");
    }
}
```

Lab Exercise 10: Abstract Factory for Operating System UI Components

Task:

Create an Abstract Factory pattern to create UI components (e.g., `Window`, `Button`) for different operating systems (e.g., `WindowsOS`, `MacOS`).

Solution:

```csharp
public interface IWindow
{
    void Render();
}

public interface IButton
{
    void Click();
}

public class WindowsWindow : IWindow
{
    public void Render()
    {
        Console.WriteLine("Rendering Windows Window");
    }
}

public class MacOSWindow : IWindow
{
    public void Render()
    {
        Console.WriteLine("Rendering MacOS Window");
    }
}
```

```
public class WindowsButton : IButton
{
    public void Click()
    {
        Console.WriteLine("Clicking Windows Button");
    }
}

public class MacOSButton : IButton
{
    public void Click()
    {
        Console.WriteLine("Clicking MacOS Button");
    }
}

public interface IUIFactory
{
    IWindow CreateWindow();
    IButton CreateButton();
}

public class WindowsUIFactory : IUIFactory
{
    public IWindow CreateWindow()
    {
        return new WindowsWindow();
    }

    public IButton CreateButton()
    {
        return new WindowsButton();
    }
}

public class MacOSUIFactory : IUIFactory
{
    public IWindow CreateWindow()
    {
        return new MacOSWindow();
    }

    public IButton CreateButton()
    {
        return new MacOSButton();
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        IUIFactory factory = new WindowsUIFactory();
        var window = factory.CreateWindow();
        var button = factory.CreateButton();

        window.Render();
        button.Click();

        factory = new MacOSUIFactory();
        window = factory.CreateWindow();
        button = factory.CreateButton();

        window.Render();
        button.Click();
    }
}
```

Lab Exercise 11: Builder Pattern for Computer Assembly

Task:

Implement a Builder pattern to assemble a `Computer` with different configurations (e.g., `GamingPC`, `OfficePC`).

Solution:

```csharp
public class Computer
{
    public string CPU { get; set; }
    public string GPU { get; set; }
    public string RAM { get; set; }
    public string Storage { get; set; }

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

public interface IComputerBuilder
{
    void BuildCPU();
    void BuildGPU();
    void BuildRAM();
    void BuildStorage();
    Computer GetComputer();
}

public class GamingPCBuilder : IComputerBuilder
{
    private Computer _computer = new Computer();

    public void BuildCPU()
    {
        _computer.CPU = "Intel i9";
    }

    public void BuildGPU()
    {
        _computer.GPU = "NVIDIA RTX 3080";
    }

    public void BuildRAM()
    {
        _computer.RAM = "32GB";
    }

    public void BuildStorage()
    {
        _computer.Storage = "1TB SSD";
    }

    public Computer GetComputer()
    {
        return _computer;
    }
}

public class OfficePCBuilder : IComputerBuilder
{
    private Computer _computer = new Computer();

    public void BuildCPU()
    {
        _computer.CPU = "Intel i5";
    }

    public void BuildGPU()
    {
        _computer.GPU = "Integrated Graphics";
    }

    public void BuildRAM()
    {
        _computer.RAM = "16GB";
    }

    public void BuildStorage()
    {
        _computer.Storage = "512GB SSD";
    }

    public Computer GetComputer()
    {
        return _computer;
    }
}

public class Director
{
    private IComputerBuilder _builder;

    public void SetBuilder(IComputerBuilder builder)
    {
        _builder = builder;
    }

    public void BuildComputer()
    {
        _builder.BuildCPU();
        _builder.BuildGPU();
        _builder.BuildRAM();
        _builder.BuildStorage();
    }

    public Computer GetComputer()
    {
        return _builder.GetComputer();
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        Director director = new Director();

        IComputerBuilder gamingPCBuilder = new GamingPCBuilder();
        director.SetBuilder(gamingPCBuilder);
        director.BuildComputer();

        Computer gamingPC = director.GetComputer();
        gamingPC.ShowSpecs();

        IComputerBuilder officePCBuilder = new OfficePCBuilder();
        director.SetBuilder(officePCBuilder);
        director.BuildComputer();

        Computer officePC = director.GetComputer();
        officePC.ShowSpecs();
    }
}
```

Lab Exercise 12: Prototype Pattern for Cloning Books

Task:

Create a Prototype pattern to clone `Book` objects with properties such as `Title`, `Author`, and `ISBN`.

Solution:

```csharp
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }
    public string ISBN { get; set; }

    public Book Clone()
    {
        return (Book)this.MemberwiseClone();
    }

    public void ShowDetails()
    {
        Console.WriteLine($"Title: {Title}, Author: {Author}, ISBN: {ISBN}");
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        Book book1 = new Book { Title = "Design Patterns", Author = "Erich Gamma", ISBN = "978-0201633610" };
        Book book2 = book1.Clone();
        book2.Title = "Refactoring";

        book1.ShowDetails();
        book2.ShowDetails();
    }
}
```

Lab Exercise 13: Singleton with Double-Checked Locking

Task:

Implement a Singleton class with double-checked locking to improve performance.

Solution:

```csharp
public class Singleton
{
    private static Singleton _instance;
    private static readonly object _lock = new object();

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            if (_instance == null)
            {
                lock (_lock)
                {
                    if (_instance == null)
                    {
                        _instance = new Singleton();
                    }
                }
            }
            return _instance;
        }
    }

    public void ShowMessage()
    {
        Console.WriteLine("Double-checked locking Singleton instance created!");
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        Singleton instance = Singleton.Instance;
        instance.ShowMessage();
    }
}
```

Lab Exercise 14: Factory Method for Document Readers

Task:

Create a Factory Method pattern for different document readers (`PDFReader`, `WordReader`).

Solution:

```csharp
public interface IDocumentReader
{
    void Open(string filePath);
}

public class PDFReader : IDocumentReader
{
    public void Open(string filePath)
    {
        Console.WriteLine($"Opening PDF document: {filePath}");
    }
}

public class WordReader : IDocumentReader
{
    public void Open(string filePath)
    {
        Console.WriteLine($"Opening Word document: {filePath}");
    }
}

public abstract class DocumentReaderCreator



{
    public abstract IDocumentReader FactoryMethod();

    public void OpenDocument(string filePath)
    {
        var reader = FactoryMethod();
        reader.Open(filePath);
    }
}

public class PDFReaderCreator : DocumentReaderCreator
{
    public override IDocumentReader FactoryMethod()
    {
        return new PDFReader();
    }
}
```

```
public class WordReaderCreator : DocumentReaderCreator
{
    public override IDocumentReader FactoryMethod()
    {
        return new WordReader();
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        DocumentReaderCreator creator = new PDFReaderCreator();
        creator.OpenDocument("file.pdf");

        creator = new WordReaderCreator();
        creator.OpenDocument("file.docx");
    }
}
```

Lab Exercise 15: Abstract Factory for Cross-Platform UI Components

Task:

Create an Abstract Factory pattern to create UI components (`Menu`, `Toolbar`) for different platforms (`Windows`, `Linux`).

Solution:

```csharp
public interface IMenu
```

```
{
    void Render();
}

public interface IToolbar
{
    void Render();
}

public class WindowsMenu : IMenu
{
    public void Render()
    {
        Console.WriteLine("Rendering Windows Menu");
    }
}

public class LinuxMenu : IMenu
{
    public void Render()
    {
        Console.WriteLine("Rendering Linux Menu");
    }
}

public class WindowsToolbar : IToolbar
{
    public void Render()
    {
        Console.WriteLine("Rendering Windows Toolbar");
    }
```

```
}

public class LinuxToolbar : IToolbar
{
    public void Render()
    {
        Console.WriteLine("Rendering Linux Toolbar");
    }
}

public interface IUIFactory
{
    IMenu CreateMenu();
    IToolbar CreateToolbar();
}

public class WindowsUIFactory : IUIFactory
{
    public IMenu CreateMenu()
    {
        return new WindowsMenu();
    }

    public IToolbar CreateToolbar()
    {
        return new WindowsToolbar();
    }
}

public class LinuxUIFactory : IUIFactory
{
    public IMenu CreateMenu()
    {
        return new LinuxMenu();
    }

    public IToolbar CreateToolbar()
    {
        return new LinuxToolbar();
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        IUIFactory factory = new WindowsUIFactory();
        var menu = factory.CreateMenu();
        var toolbar = factory.CreateToolbar();

        menu.Render();
        toolbar.Render();

        factory = new LinuxUIFactory();
        menu = factory.CreateMenu();
        toolbar = factory.CreateToolbar();

        menu.Render();
        toolbar.Render();
    }
}
```

~~~

Lab Exercise 16: Builder Pattern for Vehicle Construction

Task:

Implement a Builder pattern to construct different types of vehicles (`Car`, `Motorcycle`).

Solution:

```csharp
public class Vehicle
{
    public string Engine { get; set; }
    public string Wheels { get; set; }
    public string Frame { get; set; }

    public void ShowSpecs()
    {
        Console.WriteLine($"Engine: {Engine}, Wheels: {Wheels}, Frame: {Frame}");
    }
}

public interface IVehicleBuilder
{
    void BuildEngine();
    void BuildWheels();
    void BuildFrame();
    Vehicle GetVehicle();
}

public class CarBuilder : IVehicleBuilder
{
    private Vehicle _vehicle = new Vehicle();
~~~

```
    public void BuildEngine()
    {
        _vehicle.Engine = "V8";
    }

    public void BuildWheels()
    {
        _vehicle.Wheels = "4";
    }

    public void BuildFrame()
    {
        _vehicle.Frame = "Car Frame";
    }

    public Vehicle GetVehicle()
    {
        return _vehicle;
    }
}

public class MotorcycleBuilder : IVehicleBuilder
{
    private Vehicle _vehicle = new Vehicle();

    public void BuildEngine()
    {
        _vehicle.Engine = "500cc";
    }
    public void BuildWheels()
    {
        _vehicle.Wheels = "2";
    }

    public void BuildFrame()
    {
        _vehicle.Frame = "Motorcycle Frame";
    }

    public Vehicle GetVehicle()
    {
        return _vehicle;
    }
}

public class Director
{
    private IVehicleBuilder _builder;

    public void SetBuilder(IVehicleBuilder builder)
    {
        _builder = builder;
    }

    public void BuildVehicle()
    {
        _builder.BuildEngine();
        _builder.BuildWheels();
        _builder.BuildFrame();
    }
    public Vehicle GetVehicle()
    {
        return _builder.GetVehicle();
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        Director director = new Director();

        IVehicleBuilder carBuilder = new CarBuilder();
        director.SetBuilder(carBuilder);
        director.BuildVehicle();

        Vehicle car = director.GetVehicle();
        car.ShowSpecs();

        IVehicleBuilder motorcycleBuilder = new MotorcycleBuilder();
        director.SetBuilder(motorcycleBuilder);
        director.BuildVehicle();

        Vehicle motorcycle = director.GetVehicle();
        motorcycle.ShowSpecs();
    }
}
```

Lab Exercise 17: Prototype Pattern for Cloning Employees

Task:

Create a Prototype pattern for cloning `Employee` objects with properties such as `Name`, `Position`, and `Salary`.

Solution:

```csharp
public class Employee
{
    public string Name { get; set; }
    public string Position { get; set; }
    public double Salary { get; set; }

    public Employee Clone()
    {
        return (Employee)this.MemberwiseClone();
    }

    public void ShowDetails()
    {
        Console.WriteLine($"Name: {Name}, Position: {Position}, Salary: {Salary}");
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        Employee emp1 = new Employee { Name = "John Doe", Position = "Manager", Salary = 75000 };
        Employee emp2 = emp1.Clone();
        emp2.Name = "Jane Doe";

        emp1.ShowDetails();
        emp2.ShowDetails();
    }
}
```

Lab Exercise 18: Singleton with Reflection Protection

Task:

Modify the Singleton class to protect against instantiation via reflection.

Solution:

```csharp
public class Singleton
{
    private static Singleton _instance;
    private static readonly object _lock = new object();

    private Singleton()
    {
        if (_instance != null)
        {
            throw new InvalidOperationException("Cannot create another instance of Singleton");
        }
    }

    public static Singleton Instance
    {
        get
        {
            lock (_lock)
            {
                if (_instance == null)
                {
                    _instance = new Singleton();
                }
                return _instance;
            }
        }
    }

    public void ShowMessage()
    {
        Console.WriteLine("Singleton instance created!");
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        Singleton instance = Singleton.Instance;
        instance.ShowMessage();

        // Singleton instantiation via reflection will throw an exception
    }
}
```

Lab Exercise 19: Factory Method for Shape Creation

Task:

Create a Factory Method pattern to instantiate different shapes (`Circle`, `Square`).

Solution:

```csharp
public interface IShape
{
    void Draw();
}

public class Circle : IShape
{
    public void Draw()
    {
        Console.WriteLine("Drawing Circle");
    }
}

public class Square : IShape
{
    public void Draw()
    {
        Console.WriteLine("Drawing Square");
    }
}

public abstract class ShapeCreator
{
    public abstract IShape FactoryMethod();

    public void DrawShape()
    {
        var shape = FactoryMethod();
        shape.Draw();
    }
}

public class CircleCreator : ShapeCreator
{
    public override IShape FactoryMethod()
    {
        return new Circle();
    }
}

public class SquareCreator : ShapeCreator
{
    public override IShape FactoryMethod()
    {
        return new Square();
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        ShapeCreator creator = new CircleCreator();
        creator.DrawShape();

        creator = new SquareCreator();
        creator.DrawShape();
    }
}
```

Lab Exercise 20: Abstract Factory for Database Connections

Task:

Create an Abstract Factory pattern to create database connections (`SQLConnection`, `OracleConnection`) and commands (`SQLCommand`, `OracleCommand`).

Solution:

```csharp
public interface IDbConnection
{
    void Connect();
}

public interface IDbCommand
{
    void Execute();
}

public class SQLConnection : IDbConnection
{
    public void Connect()
    {
        Console.WriteLine("Connecting to SQL Server");
    }
}

public class OracleConnection : IDbConnection
```

```
{
    public void Connect()
    {
        Console.WriteLine("Connecting to Oracle Database");
    }
}

public class SQLCommand : IDbCommand
{
    public void Execute()
    {
        Console.WriteLine("Executing SQL Command");
    }
}

public class OracleCommand : IDbCommand
{
    public void Execute()
    {
        Console.WriteLine("Executing Oracle Command");
    }
}

public interface IDatabaseFactory
{
    IDbConnection CreateConnection();
    IDbCommand CreateCommand();
}

public class SQLDatabaseFactory : IDatabaseFactory
{
    public IDbConnection CreateConnection()
    {
        return new SQLConnection();
    }

    public IDbCommand CreateCommand()
    {
        return new SQLCommand();
    }
}

public class OracleDatabaseFactory : IDatabaseFactory
{
    public IDbConnection CreateConnection()
    {
        return new OracleConnection();
    }

    public IDbCommand CreateCommand()
    {
        return new OracleCommand();
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        IDatabaseFactory factory = new SQLDatabaseFactory();
        var connection = factory
.CreateConnection();
        var command = factory.CreateCommand();

        connection.Connect();
        command.Execute();

        factory = new OracleDatabaseFactory();
        connection = factory.CreateConnection();
        command = factory.CreateCommand();

        connection.Connect();
        command.Execute();
    }
}
```