1. Single Responsibility Principle (SRP)

```
using System;

// Class for managing user data
public class UserManager {
    public string UserName { get; set; }

    // Manages user data storage
    public void SaveUser() {
        Console.WriteLine("Saving user to the database.");
    }
}

// Class for handling user logging
public class Logger {
    // Responsible for logging user actions
    public void LogUserAction() {
        Console.WriteLine("Logging user action.");
    }
}

class Program {
    static void Main(string[] args) {
        // **SRP Example**: Separated responsibilities into distinct classes
        UserManager userManager = new UserManager { UserName = "John" };
        Logger logger = new Logger();

        userManager.SaveUser();  // Saving user
        logger.LogUserAction();  // Logging user action
    }
}
```

- **Explanation**: The `UserManager` class is responsible only for user data, while the `Logger` class handles logging. This ensures each class has only one responsibility, adhering to SRP.

---

2. Open/Closed Principle (OCP)

```
using System;

// Open/Closed: The classes are open for extension but closed for modification.
public interface IShape {
    double CalculateArea();  // Open for extension
}

public class Circle : IShape {
    public double Radius { get; set; }

    public double CalculateArea() {
        return Math.PI * Radius * Radius; // Circle area
    }
}

public class Rectangle : IShape {
    public double Width { get; set; }
    public double Height { get; set; }

    public double CalculateArea() {
        return Width * Height;  // Rectangle area
    }
}

public class AreaCalculator {
    public double CalculateArea(IShape shape) {
        return shape.CalculateArea();
    }
}

class Program {
    static void Main(string[] args) {
        // **OCP Example**: Adding new shapes without modifying the AreaCalculator class.
        IShape circle = new Circle { Radius = 5 };
        IShape rectangle = new Rectangle { Width = 4, Height = 6 };

        AreaCalculator areaCalculator = new AreaCalculator();
        Console.WriteLine("Circle area: " + areaCalculator.CalculateArea(circle));
        Console.WriteLine("Rectangle area: " + areaCalculator.CalculateArea(rectangle));
    }
}
```

- **Explanation**: The `AreaCalculator` is closed for modification but open for extension. New shapes can be added (like `Circle`, `Rectangle`) without changing the existing class.

---

3. Liskov Substitution Principle (LSP)

```
using System;

// Base class Bird
public class Bird {
    public virtual void Move() {
        Console.WriteLine("Bird is flying.");
    }
}

// Derived class Sparrow
public class Sparrow : Bird {
    public override void Move() {
        Console.WriteLine("Sparrow is flying.");
    }
}

// Derived class Ostrich
public class Ostrich : Bird {
    public override void Move() {
        Console.WriteLine("Ostrich is running.");
    }
}

class Program {
    static void Main(string[] args) {
        // **LSP Example**: Substituting Bird class with its subclasses without breaking behavior
        Bird sparrow = new Sparrow();
        Bird ostrich = new Ostrich();

        sparrow.Move();  // Sparrow should fly
        ostrich.Move();  // Ostrich should run
    }
}
```

- **Explanation**: `Sparrow` and `Ostrich` are derived from `Bird` and both override the `Move()` method in a way that maintains the behavior expected of the base class.

---

4. Interface Segregation Principle (ISP)

```
using System;

// Interface for workers who can work
public interface IWorker {
    void Work();
}

// Interface for workers who can rest
public interface IRestable {
    void Rest();
}

public class Worker : IWorker, IRestable {
    public void Work() {
        Console.WriteLine("Worker is working.");
    }

    public void Rest() {
        Console.WriteLine("Worker is resting.");
    }
}

public class Robot : IWorker {
    public void Work() {
        Console.WriteLine("Robot is working.");
    }
}

class Program {
    static void Main(string[] args) {
        // **ISP Example**: Separated interfaces so Robot doesn't need to implement Rest
        IWorker worker = new Worker();
        worker.Work();

        IRestable restableWorker = new Worker();
        restableWorker.Rest();

        IWorker robot = new Robot();
        robot.Work();  // Robot doesn't need to rest
    }
}
```

- **Explanation**: The `IWorker` and `IRestable` interfaces are separated so that a `Robot` only needs to implement `IWorker`, avoiding unnecessary methods.

---

5. Dependency Inversion Principle (DIP)

```
using System;

// High-level interface for devices
public interface IDevice {
    void TurnOn();
    void TurnOff();
}

// Concrete implementation of a device: LightBulb
public class LightBulb : IDevice {
    public void TurnOn() {
        Console.WriteLine("LightBulb is now ON.");
    }

    public void TurnOff() {
        Console.WriteLine("LightBulb is now OFF.");
    }
}

// Switch class depends on IDevice abstraction, not the concrete class.
public class Switch {
    private readonly IDevice _device;

    public Switch(IDevice device) {
        _device = device;
    }

    public void Operate() {
        _device.TurnOn();
    }
}

class Program {
    static void Main(string[] args) {
        // **DIP Example**: Switch depends on an abstraction (IDevice) instead of a concrete class
        IDevice lightBulb = new LightBulb();
        Switch lightSwitch = new Switch(lightBulb);
        lightSwitch.Operate();  // Turns on the LightBulb
    }
}
```

- **Explanation**: `Switch` depends on the `IDevice` abstraction, not a concrete class like `LightBulb`. This makes it easier to replace or extend the device types.

How to Run:

1. Copy each of the code blocks into separate `.cs` files (e.g., `SRPExample.cs`, `OCPExample.cs`, etc.).
2. Compile and run each example using a C# compiler, such as Visual Studio, or via the command line using `csc <filename.cs>`.
3. The program will demonstrate one SOLID principle at a time.

This way, each principle is showcased in isolation, making it easier to understand how it works in a real-world context.