Exercise 1: Polymorphism and Syntax of Interface

Lab Exercise:

1. Define an interface `IShape` with a method `Draw()`.

2. Implement this interface in two classes: `Circle` and `Rectangle`.

3. Demonstrate polymorphism by calling the `Draw()` method on an array of `IShape` objects containing both `Circle` and `Rectangle`.

Solution:

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

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

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

```
class Program
{
    static void Main(string[] args)
    {
        IShape[] shapes = new IShape[] { new Circle(), new Rectangle() };
        foreach (var shape in shapes)
        {
            shape.Draw();
        }
    }
}
```

Exercise 2: Explicit Implementation & Casting

Lab Exercise:

1. Create an interface `IPrintable` with a method `Print()`.
2. Create a class `Document` that implements `IPrintable` with explicit interface implementation.
3. Demonstrate casting the object to the interface to call the `Print()` method.

Solution:

```csharp
public interface IPrintable
{
    void Print();
}

public class Document : IPrintable
{
    void IPrintable.Print()
    {
        Console.WriteLine("Printing document");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Document doc = new Document();
        // Cannot call doc.Print() directly since it's explicit implementation
        IPrintable printable = doc;
        printable.Print(); // Printing document
    }
}
```

Exercise 3: Types of Interfaces

Lab Exercise:

1. Create two interfaces `IDrawable` and `IPrintable`.

2. Implement both interfaces in a class `Photo`.

3. Demonstrate using a class that implements multiple interfaces.

Solution:

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

public interface IPrintable
```

```
{
    void Print();
}

public class Photo : IDrawable, IPrintable
{
    public void Draw()
    {
        Console.WriteLine("Drawing the photo");
    }

    public void Print()
    {
        Console.WriteLine("Printing the photo");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Photo photo = new Photo();
        photo.Draw();
        photo.Print();
    }
}
```

Exercise 4: Method Overloading

Lab Exercise:

1. Create a class `MathOperations` with overloaded methods `Add()`.

2. Provide overloads for adding two integers, two doubles, and three integers.

3. Demonstrate calling each overloaded method.

Solution:

```csharp
public class MathOperations
{
    public int Add(int a, int b)
    {
        return a + b;
    }

    public double Add(double a, double b)
    {
        return a + b;
    }

    public int Add(int a, int b, int c)
    {
        return a + b + c;
    }
}

class Program
{
    static void Main(string[] args)
    {
        MathOperations math = new MathOperations();

        Console.WriteLine(math.Add(10, 20));         // 30
        Console.WriteLine(math.Add(10.5, 20.3));     // 30.8
        Console.WriteLine(math.Add(10, 20, 30));     // 60
    }
}
```

Exercise 5: Method Overriding

Lab Exercise:

1. Create a base class `Animal` with a virtual method `Speak()`.

2. Create a derived class `Dog` that overrides the `Speak()` method.

3. Demonstrate method overriding by calling the `Speak()` method on both `Animal` and `Dog` objects.

Solution:

```csharp
public class Animal
{
    public virtual void Speak()
    {
        Console.WriteLine("Animal makes a sound");
    }
}

public class Dog : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Dog barks");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Animal animal = new Animal();
        animal.Speak();  // Animal makes a sound

        Dog dog = new Dog();
        dog.Speak();  // Dog barks

        Animal anotherDog = new Dog();
        anotherDog.Speak();  // Dog barks (runtime polymorphism)
    }
}
```

Exercise 6: Virtual Keyword

Lab Exercise:

1. Create a base class `BaseClass` with a virtual method `Display()`.
2. Create a derived class `DerivedClass` that overrides the `Display()` method.
3. Demonstrate the use of the `virtual` and `override` keywords.

Solution:

```csharp
public class BaseClass
{
    public virtual void Display()
    {
        Console.WriteLine("BaseClass Display");
    }
}

public class DerivedClass : BaseClass
{
    public override void Display()
    {
        Console.WriteLine("DerivedClass Display");
    }
}

class Program
{
    static void Main(string[] args)
    {
        BaseClass baseObj = new BaseClass();
        baseObj.Display();  // BaseClass Display

        DerivedClass derivedObj = new DerivedClass();
        derivedObj.Display();  // DerivedClass Display

        BaseClass polymorphicObj = new DerivedClass();
        polymorphicObj.Display();  // DerivedClass Display (runtime polymorphism)
    }
}
```

Exercise 7: Late Binding vs Early Binding

Lab Exercise:

1. Create a base class `Printer` with a non-virtual method `Print()`.

2. Create a derived class `LaserPrinter` that hides the `Print()` method.

3. Demonstrate early binding by calling the `Print()` method on a `Printer` reference and late binding by using virtual/override.

Solution:

```csharp
public class Printer
{
    public void Print()
    {
        Console.WriteLine("Printing from Printer");
    }
}

public class LaserPrinter : Printer
{
    public new void Print()
    {
        Console.WriteLine("Printing from LaserPrinter");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Printer printer = new Printer();
        printer.Print();  // Printing from Printer (early binding)

        LaserPrinter laserPrinter = new LaserPrinter();
        laserPrinter.Print();  // Printing from LaserPrinter (early binding)
```

        Printer polymorphicPrinter = new LaserPrinter();
        polymorphicPrinter.Print(); // Printing from Printer (early binding)
    }
}
~~~

Exercise 8: Runtime Polymorphism

Lab Exercise:

1. Create a base class `Employee` with a virtual method `CalculateSalary()`.
2. Create derived classes `Manager` and `Developer` that override `CalculateSalary()`.
3. Demonstrate runtime polymorphism by calling `CalculateSalary()` on different types of `Employee`.

Solution:

```csharp
public class Employee
{
    public virtual void CalculateSalary()
    {
        Console.WriteLine("Calculating salary for Employee");
    }
}

public class Manager : Employee
{
    public override void CalculateSalary()
    {
        Console.WriteLine("Calculating salary for Manager");
    }
}
~~~

```
public class Developer : Employee
{
    public override void CalculateSalary()
    {
        Console.WriteLine("Calculating salary for Developer");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Employee emp1 = new Manager();
        emp1.CalculateSalary();  // Calculating salary for Manager

        Employee emp2 = new Developer();
        emp2.CalculateSalary();  // Calculating salary for Developer
    }
}
```

Exercise 9: Façade Pattern

Lab Exercise:

1. Create a façade class `HomeTheaterFacade` that wraps the complexity of `DVDPlayer`, `Amplifier`, and `Projector` classes.

2. Provide a simple interface in `HomeTheaterFacade` to start and stop the movie.

3. Demonstrate using the façade to control the home theater system.

Solution:

```csharp
public class DVDPlayer
{
    public void On() => Console.WriteLine("DVD Player On");
    public void Play() => Console.WriteLine("DVD Player Playing");
    public void Off() => Console.WriteLine("DVD Player Off");
}

public class Amplifier
{
    public void On() => Console.WriteLine("Amplifier On");
    public void SetVolume(int level) => Console.WriteLine($"Amplifier Volume set to {level}");
    public void Off() => Console.WriteLine("Amplifier Off");
}

public class Projector
{
    public void On() => Console.WriteLine("Projector On");
    public void SetInput(string source) => Console.WriteLine($"Projector input set to {source}");
    public void Off() => Console.WriteLine("Projector Off");
}

public class HomeTheaterFacade
{
    private DVDPlayer dvdPlayer;
    private Amplifier amplifier;
    private Projector projector;

    public HomeTheaterFacade(DVDPlayer dvd, Amplifier amp, Projector proj)
    {
        dvdPlayer = dvd;
        amplifier = amp;
        projector = proj;
    }

    public void WatchMovie()
    {
        Console.WriteLine("Starting Movie...");
        dvdPlayer.On();
        dvdPlayer.Play();
        amplifier.On();
        amplifier.Set

Volume(5);
        projector.On();
        projector.SetInput("DVD");
    }

    public void EndMovie()
    {
        Console.WriteLine("Stopping Movie...");
        dvdPlayer.Off();
        amplifier.Off();
        projector.Off();
    }
}

class Program
{
    static void Main(string[] args)
    {
        DVDPlayer dvdPlayer = new DVDPlayer();
```

        Amplifier amplifier = new Amplifier();
        Projector projector = new Projector();

        HomeTheaterFacade homeTheater = new HomeTheaterFacade(dvdPlayer, amplifier, projector);
        homeTheater.WatchMovie();
        homeTheater.EndMovie();
    }
}
~~~

Exercise 10: Interface Segregation Principle

Lab Exercise:

1. Create interfaces `IReadable` and `IWritable` with methods `Read()` and `Write()`.

2. Implement these interfaces in a class `FileHandler`.

3. Demonstrate the use of Interface Segregation Principle by using different classes that implement different combinations of these interfaces.

Solution:

```csharp
public interface IReadable
{
    void Read();
}

public interface IWritable
{
    void Write();
}

public class FileHandler : IReadable, IWritable
~~~

```
{
    public void Read()
    {
        Console.WriteLine("Reading from file");
    }

    public void Write()
    {
        Console.WriteLine("Writing to file");
    }
}

public class ReadOnlyHandler : IReadable
{
    public void Read()
    {
        Console.WriteLine("Reading from file (read-only)");
    }
}

public class WriteOnlyHandler : IWritable
{
    public void Write()
    {
        Console.WriteLine("Writing to file (write-only)");
    }
}

class Program
{
    static void Main(string[] args)
    {
        IReadable reader = new ReadOnlyHandler();
        reader.Read();

        IWritable writer = new WriteOnlyHandler();
        writer.Write();

        FileHandler fileHandler = new FileHandler();
        fileHandler.Read();
        fileHandler.Write();
    }
}
```