Abstract and Interface

Abstraction:

Abstraction is the process of hiding the implementation details and showing only the functionality to the user. It helps in reducing complexity by allowing the user to interact with an object at a high level without needing to understand the details of its internal workings.

Abstract Classes in C#

An abstract class is a class that cannot be instantiated on its own and is designed to be inherited by other classes. It can have abstract methods (methods without a body) that must be implemented by derived classes.

1. The abstract keyword is used to create an abstract class.

```
public abstract class Animal
{
    public abstract void MakeSound(); // Abstract method
}
```

2. An abstract class must be inherited and its abstract members implemented.

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

3. An abstract class can only be used as a base class; it cannot be sealed.

```
public abstract class BaseClass
{
    public abstract void DoSomething();
}
// Cannot be sealed and abstract
// sealed class means it nIn C#, a sealed class is one that cannot be inherited.
// public sealed abstract class SealedAbstractClass // Error
```

4. An abstract class can have abstract and non-abstract members.

```
public abstract class Vehicle
{
    public abstract void StartEngine(); // Abstract method

    public void StopEngine() // Non-abstract method
    {
        Console.WriteLine("Engine stopped.");
    }
}
```

5. Inheriting an abstract class can be done in two ways:
   - Implement all abstract methods:

```
public class Car : Vehicle
{
    public override void StartEngine()
    {
        Console.WriteLine("Car engine started.");
    }
}
```

   - Declare the class as abstract if not implementing all methods:

```
public abstract class Motorcycle : Vehicle
{
    // Not implementing StartEngine, so Motorcycle is abstract
}
```

Full code:

```
using System;

abstract class Animal
{
    // Abstract method (does not have a body)
    public abstract void MakeSound();

    // Regular method
    public void Sleep()
    {
        Console.WriteLine("Sleeping...");
    }
}

class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Woof! Woof!");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Dog dog = new Dog();
        dog.MakeSound(); // Outputs: Woof! Woof!
        dog.Sleep();     // Outputs: Sleeping...
    }
}
```

Interfaces in C#

1. Interfaces have only methods with no bodies.

```
public interface IShape
{
    void Draw(); // Method without body
}
```

2. Interface members are public by default, and you cannot use public modifier.

```
public interface IAnimal
{
    void Speak(); // Implicitly public
}
```

3. Interfaces cannot have fields.

```
public interface ICar
{
    // int speed; // Error: Interfaces cannot contain fields
    void Drive();
}
```

4. A class inheriting an interface must implement all its methods.

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

5. A class can inherit multiple interfaces.

```
public interface IMoveable
{
    void Move();
}

public interface IFlyable
{
    void Fly();
}

public class Bird : IMoveable, IFlyable
{
    public void Move() { Console.WriteLine("Bird moves."); }
    public void Fly() { Console.WriteLine("Bird flies."); }
}
```

6. In multilevel interfaces, a class must implement all methods. Here If IB interface inherit IA interface and finally Program class inherit IB interface then program class needs to implement all the methods from all interface.

```
using System;

interface IA
{
    void F1();
}

interface IB : IA
{
    void F2();
}

public class Program : IB
{
    public void F1()
    {
        Console.WriteLine("F1 implemented.");
    }

    public void F2()
    {
        Console.WriteLine("F2 implemented.");
    }

    static void Main(string[] args)
    {
        Program program = new Program();
        program.F1(); // Outputs "F1 implemented."
        program.F2(); // Outputs "F2 implemented."
    }
}
```

7. Type Casting Interface

   Need to apply when in two interface functions name are same, then we need to mention in which time which method should be called by using Type casting interface

```
using System;

interface IA
{
    void Fun();
}

interface IB
{
    void Fun();
}

public class Program : IA, IB
{
    void IA.Fun(){ Console.WriteLine("This function is for IA interface");}
    // can use access modifier like public in the begining and
    // need to use the interface name before the function name like IA.F1()

    void IB.Fun(){ Console.WriteLine("This function is for IB interface");}

    public static void Main(string[] args)
    {
        Program pg = new Program();
        ((IA)pg).Fun(); // Explicit interface - interface name. object name. function name
        ((IB)pg).Fun();
    }
}
```

Full interface Code

```
using System;

interface IAnimal
{
    void MakeSound();
    void Sleep();
}

class Dog : IAnimal
{
    public void MakeSound()
    {
        Console.WriteLine("Woof! Woof!");
    }

    public void Sleep()
    {
        Console.WriteLine("Sleeping...");
    }
}

class Program
{
    static void Main(string[] args)
    {
        IAnimal dog = new Dog();
        dog.MakeSound(); // Outputs: Woof! Woof!
        dog.Sleep();     // Outputs: Sleeping...
    }
}
```

Differences Between Interface and Abstract Class

Feature	Interface	Abstract Class
Method Implementation	No method bodies; only declarations	Can have both abstract and non-abstract methods
Access Modifiers	Members are implicitly public	Members can have access modifiers
Fields	Cannot contain fields	Can contain fields
Multiple Inheritance	A class can implement multiple interfaces	A class can inherit only one abstract class
Constructors	Cannot have constructors	Can have constructors