What is an Interface in C#?

Definition:
An **interface** in C# is a contract that defines a set of methods and properties that a class must implement. It does not contain any implementation itself; it only provides the signatures of the methods.

Purpose:
Interfaces allow you to define the **"what"** without the **"how."** A class that implements an interface must provide concrete implementations of the methods and properties defined by the interface.

Example:

```
public interface IVehicle

{

        void Start();

        void Stop();

        int Speed { get; set; }

}
```

Key Characteristics of Interfaces

- **No Implementation:**
  Interfaces contain method signatures, but no method bodies. This means they cannot define any functionality themselves.
- **Multiple Inheritance:**
  A class can implement multiple interfaces, allowing you to **"inherit"** behavior from more than one source. In contrast, a class can only inherit from one base class.
- **Used for Abstraction:**
  Interfaces are used to achieve **abstraction**, where you only expose certain methods to the user without revealing the underlying implementation.

Implementing an Interface

- **How to Implement:**
  A class that implements an interface must provide definitions for all the methods and properties declared in the interface. If the class does not, it must be marked as abstract.
- **Example:**

```
public class Car : IVehicle

{           public int Speed { get; set; }

            public void Start()

            {

            Console.WriteLine("Car is starting.");

            }

            public void Stop()

            {

            Console.WriteLine("Car is stopping.");

            }

}
```

Multiple Interface Implementation:

A class can implement multiple interfaces. For example:

```
public interface IFlyable

{

        void Fly();

}

public class FlyingCar : IVehicle, IFlyable

{

        public int Speed { get; set; }

        public void Start() { Console.WriteLine("Flying Car is starting."); }

        public void Stop() { Console.WriteLine("Flying Car is stopping."); }

        public void Fly() { Console.WriteLine("Flying Car is flying."); }

}
```

Interface Properties

- **Properties in Interfaces:**
  Interfaces can define properties, but like methods, they only provide the signature and not the implementation:

```
public interface IEmployee

{

    string Name { get; set; }

    double Salary { get; }

}
```

Property Implementation Example:

```
public class Manager : IEmployee
{
    public string Name { get; set; }
    public double Salary { get; private set; }
    public Manager(string name, double salary)
    {
    Name = name;
    Salary = salary;
    }
}
```

Interface vs Abstract Class

- **Differences:**
    - **Multiple Inheritance:** A class can implement multiple interfaces but can only inherit from one abstract class.
    - **Methods with Bodies:** Abstract classes can have method implementations, but interfaces cannot.
    - **Fields:** Interfaces cannot have fields (variables), whereas abstract classes can.

Real-World Example of Interfaces

- **Dependency Injection:**
  Interfaces are heavily used in **dependency injection** to decouple components. For example, services in a web application are often defined using interfaces, which allows the underlying implementations to be easily swapped out.
  Example in a web app:

```
public interface IEmailService

{

    void SendEmail(string recipient, string subject, string message);

}



public class SmtpEmailService : IEmailService

{

    public void SendEmail(string recipient, string subject, string message)

    {

    // SMTP email sending logic here

    }

}
```

```
public class NotificationService
{
    private readonly IEmailService _emailService;
    public NotificationService(IEmailService emailService)
    {
    _emailService = emailService;
    }
    public void Notify(string recipient)
    {
    _emailService.SendEmail(recipient, "Welcome", "Thank you for signing up!");
    }
}
```

Best Practices

- **Use for Contracts:**
  Define interfaces when you need to specify **what** a class should do, but not **how** it should do it.
- **Favor Composition Over Inheritance:**
  When designing systems, it’s often better to use interfaces to compose functionality rather than relying on deep class inheritance hierarchies.