Interfaces in C#

In C#, an interface is a contract that defines a set of methods, properties, events, or indexers that a class or struct must implement. Interfaces provide a way to enforce that certain classes have specific behaviors without dictating how those behaviors should be implemented. Here's a detailed explanation with examples to illustrate the concept:

Defining an Interface

An interface is defined using the interface keyword, followed by the name of the interface. By convention, interface names start with an uppercase "I". Interfaces only contain the declaration of members without any implementation. Here's an example:

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

In this example, IPrintable is an interface with a single method Print. Any class that implements IPrintable must provide an implementation for the Print method.

Implementing an Interface

To implement an interface, a class must use the : InterfaceName syntax and provide definitions for all of the members defined in the interface. Here's an example:

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

In this example, the Document class implements the IPrintable interface by providing an implementation for the Print method. Any instance of Document can be treated as an IPrintable object.

Using Multiple Interfaces

C# allows a class to implement multiple interfaces. This is useful when you want a class to adhere to multiple contracts. Here’s an example:

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

public interface IScannable
{
    void Scan();
}

public class MultiFunctionPrinter : IPrintable, IScannable
{
    public void Print()
    {
        Console.WriteLine("Printing from MultiFunctionPrinter...");
    }

    public void Scan()
    {
        Console.WriteLine("Scanning from MultiFunctionPrinter...");
    }
}
```

In this example, MultiFunctionPrinter implements both IPrintable and IScannable, meaning it must provide implementations for both the Print and Scan methods.

Interface Inheritance

Interfaces can inherit from other interfaces, allowing for more specialized behavior. Here's how interface inheritance works:

```
public interface IMachine
{
   void Start();
   void Stop();
}

public interface ICopier : IMachine
{
   void Copy();
}
public class CopierMachine : ICopier
{
   public void Start()
   {
      Console.WriteLine("Copier machine starting...");
   }
   public void Stop()
   {
      Console.WriteLine("Copier machine stopping...");
   }
   public void Copy()
   {
      Console.WriteLine("Copying document...");
   }
}
```

In this example, ICopier inherits from IMachine, so any class that implements ICopier must also implement the members of IMachine. The CopierMachine class provides implementations for all methods inherited from both IMachine and ICopier.

Explicit Interface Implementation

Sometimes, a class might implement multiple interfaces that have members with the same name. To avoid conflicts, you can use explicit interface implementation, which involves qualifying the member with the interface name:

```
public interface IReadable
{
  void Print();
}

public interface IWritable
{
  void Print();
}

public class ReadWriteFile : IReadable, IWritable
{
  void IReadable.Print()
  {
    Console.WriteLine("Reading from file...");
  }

  void IWritable.Print()
  {
    Console.WriteLine("Writing to file...");
  }
}
```

In this example, ReadWriteFile implements both IReadable and IWritable, which both have a Print method. By using explicit implementation, the class specifies which method belongs to which interface.

Interface Properties and Indexers

Interfaces can also include properties and indexers. Here's an example of an interface with a property:

```
public interface IDevice
{
   string Model { get; set; }
}

public class Smartphone : IDevice
{
   public string Model { get; set; }
}
```

In this example, IDevice includes a Model property, which Smartphone implements. Any instance of Smartphone will have a Model property that gets or sets the device model.

Practical Example: Interface with Method, Property

```
public interface IAccount
{
   string AccountNumber { get; }
   decimal Balance { get; }

   void Deposit(decimal amount);
}
```

```
public interface IWithdrawable
{
    void Withdraw(decimal amount);
}

public interface IInterestBearing
{
    decimal InterestRate { get; set; }
    void ApplyInterest();
}

public class SavingsAccount : IAccount, IWithdrawable, IInterestBearing
{
    public string AccountNumber { get; private set; }
    public decimal Balance { get; private set; }
    public decimal InterestRate { get; set; }

    public SavingsAccount(string accountNumber, decimal initialBalance, decimal interestRate)
    {
        AccountNumber = accountNumber;
        Balance = initialBalance;
        InterestRate = interestRate;
    }

    public void Deposit(decimal amount)
    {
        Balance += amount;
        Console.WriteLine($"Deposited {amount:C} to savings account. New balance:
{Balance:C}");
```

```
    }

    public void Withdraw(decimal amount)
    {
        if (amount <= Balance)
        {
            Balance -= amount;
            Console.WriteLine($"Withdrew {amount:C} from savings account. New balance: {Balance:C}");
        }
        else
        {
            Console.WriteLine("Insufficient funds for withdrawal.");
        }
    }

    public void ApplyInterest()
    {
        var interest = Balance * InterestRate;
        Balance += interest;
        Console.WriteLine($"Applied interest of {interest:C}. New balance: {Balance:C}");
    }
}

public class CreditAccount : IAccount, IWithdrawable
{
    public string AccountNumber { get; private set; }
    public decimal Balance { get; private set; }
    public decimal CreditLimit { get; private set; }
```

```
    public CreditAccount(string accountNumber, decimal creditLimit)
    {
        AccountNumber = accountNumber;
        CreditLimit = creditLimit;
        Balance = -CreditLimit;
    }

    public void Deposit(decimal amount)
    {
        Balance += amount;
        Console.WriteLine($"Deposited {amount:C} to credit account. New balance: {Balance:C}");
    }

    void IWithdrawable.Withdraw(decimal amount)
    {
        if (Balance + amount <= CreditLimit)
        {
            Balance -= amount;
            Console.WriteLine($"Withdrew {amount:C} from credit account. New balance:
{Balance:C}");
        }
        else
        {
            Console.WriteLine("Credit limit exceeded.");
        }
    }
}
```

Another Example

```
public interface IEmployee
{
   int EmployeeId { get; }
   string Name { get; }
   void Work();
}

public interface IPatientCare
{
   void CheckVitals();
}

public interface IPrescriber
{
   void PrescribeMedication(string medication);
}
public class Doctor : IEmployee, IPatientCare, IPrescriber
{
   public int EmployeeId { get; private set; }
   public string Name { get; private set; }

   public Doctor(int employeeId, string name)
   {
      EmployeeId = employeeId;
      Name = name;
   }

   public void Work()
```

```
    {
        Console.WriteLine($"{Name} is diagnosing patients.");
    }

    public void CheckVitals()
    {
        Console.WriteLine($"{Name} is checking patient vitals.");
    }

    public void PrescribeMedication(string medication)
    {
        Console.WriteLine($"{Name} prescribed {medication}.");
    }
}

public class Nurse : IEmployee, IPatientCare
{
    public int EmployeeId { get; private set; }
    public string Name { get; private set; }

    public Nurse(int employeeId, string name)
    {
        EmployeeId = employeeId;
        Name = name;
    }
    public void Work()
    {
        Console.WriteLine($"{Name} is assisting in patient care.");
    }
```

```
    public void CheckVitals()
    {
        Console.WriteLine($"{Name} is checking patient vitals.");
    }
}
```

Interface vs abstract class

Whie the functionalities looks same between interface and abstract class, there is some differences should be considered, the following are these differences:

Key Differences Between Interface and Abstract Class

1. **Method Implementation**:
    - **Interface**: Can only declare methods, properties, events, or indexers, but cannot provide any implementation (except with default interface methods in C# 8.0 and later).
    - **Abstract Class**: Can have both abstract members (declarations without implementations) and concrete members (fully implemented methods).
2. **Multiple Inheritance**:
    - **Interface**: A class can implement multiple interfaces, enabling a form of multiple inheritance.
    - **Abstract Class**: A class can inherit from only one abstract class due to the single inheritance restriction in C#. However, an abstract class can implement multiple interfaces.
3. **Fields and Constructors**:
    - **Interface**: Cannot contain fields (member variables) or constructors.
    - **Abstract Class**: Can contain fields, constants, and constructors. This allows abstract classes to maintain state.
4. **Access Modifiers**:
    - **Interface**: Members are implicitly public, and access modifiers are not allowed.
    - **Abstract Class**: Members can have different access levels, such as public, protected, internal, or private, providing more control over accessibility.

5. **Versioning**:
    - **Interface**: If you modify an interface by adding a new member, all implementing classes must be updated to implement that member. In C# 8.0 and later, default interface methods can help with backward compatibility but are still limited.
    - **Abstract Class**: If you add a new method, you can provide a default implementation, which doesn't force inheriting classes to immediately implement the new method.

When to Use an Interface

1. **Multiple Inheritance of Types**: When you need a class to inherit multiple types, interfaces are the way to go, as C# doesn't support multiple inheritance with classes.
2. **Polymorphism Without Shared Implementation**: If you only need to ensure a class implements certain methods without sharing any implementation, use an interface. Interfaces define "what" a class can do, not "how" it does it.
3. **Small, Simple Contracts**: Interfaces are ideal for defining small, focused, and loosely coupled contracts, especially for loosely coupled components in large systems.
4. **Dependency Injection and Testing**: Interfaces are often used to facilitate dependency injection and create testable code by allowing mocking or stubbing of the dependencies in unit tests.

When to Use an Abstract Class

1. **Shared Code and State**: If there is shared functionality or state that you want to provide to all derived classes, use an abstract class. Abstract classes allow you to create a common base with both implemented methods and abstract methods that subclasses must override.
2. **Providing Partial Implementations**: If you want to provide some common implementation while still enforcing certain members to be implemented by derived classes, abstract classes are ideal.
3. **More Control Over Access**: Use an abstract class when you need to control access levels for members (e.g., protected members that are accessible only within derived classes).
4. **Versioning and Extensibility**: Abstract classes are more flexible when extending functionality because you can add new non-abstract methods without breaking existing subclasses.