Интерфейсы

Определение интерфейсов

Интерфейс представляет ссылочный тип, который может определять некоторый функционал - набор методов и свойств без реализации. Затем этот функционал реализуют классы и структуры, которые применяют данные интерфейсы.

Определение интерфейса

Для определения интерфейса используется ключевое слово **interface**. Как правило, названия интерфейсов в C# начинаются с заглавной буквы **I**, например, IComparable, IEnumerable (так называемая венгерская нотация), однако это не обязательное требование, а больше стиль программирования.

Что может определять интерфейс? В целом интерфейсы могут определять следующие сущности:

- Методы
- Свойства
- Индексаторы
- События
- Статические поля и константы (начиная с версии C# 8.0)

Однако интерфейсы не могут определять нестатические переменные. Например, простейший интерфейс, который определяет все эти компоненты:

```
interface IMovable
{
    // константа
    const int minSpeed = 0;     // минимальная скорость
    // статическая переменная
    static int maxSpeed = 60;   // максимальная скорость
    // метод
    void Move();                // движение
    // свойство
    string Name { get; set; }   // название

    delegate void MoveHandler(string message);  // определение делегата для события
    // событие
    event MoveHandler MoveEvent;    // событие движения
}
```

В данном случае определен интерфейс IMovable, который представляет некоторый движущийся объект. Данный интерфейс содержит различные компоненты, которые описывают возможности движущегося объекта. То есть интерфейс описывает некоторый функционал, который должен быть у движущегося объекта.

Методы и свойства интерфейса могут не иметь реализации, в этом они сближаются с абстрактными методами и свойствами абстрактных классов. В данном случае интерфейс определяет метод Move, который будет представлять некоторое передвижение. Он не имеет реализации, не принимает никаких параметров и ничего не возвращает.

То же самое в данном случае касается свойства Name. На первый взгляд оно похоже на автоматическое свойство. Но в реальности это определение свойства в интерфейсе, которое не имеет реализации, а не автосвойство.

Еще один момент в объявлении интерфейса: если его члены - методы и свойства не имеют модификаторов доступа, но фактически по умолчанию доступ **public**, так как цель интерфейса - определение функционала для реализации его классом. Это касается также и констант и статических переменных, которые в классах и структурах по умолчанию имееют модификатор private. В интерфейсах же они имеют по умолчанию модификатор public. И например, мы могли бы обратиться к константе minSpeed и переменной maxSpeed интерфейса IMovable:

```
1static void Main(string[] args)
2{
3    Console.WriteLine(IMovable.maxSpeed);
4    Console.WriteLine(IMovable.minSpeed);
5}
```

Но также, начиная с версии C# 8.0, мы можем явно указывать модификаторы доступа у компонентов интерфейса:

```
1interface IMovable
2{
3    public const int minSpeed = 0;     // минимальная скорость
4    private static int maxSpeed = 60;   // максимальная скорость
5    public void Move();
6    protected internal string Name { get; set; }    // название
7    public delegate void MoveHandler(string message);  // определение делегата для события
8    public event MoveHandler MoveEvent;    // событие движения
9}
```

Начиная с версии C# 8.0 интерфейсы поддерживают реализацию методов и свойств по умолчанию. Это значит, что мы можем определить в интерфейсах полноценные методы и свойства, которые имеют реализацию как в обычных классах и структурах. Например, определим реализацию метода Move по умолчанию:

```
1interface IMovable
2{
3    // реализация метода по умолчанию
4    void Move()
5    {
6        Console.WriteLine("Walking");
7    }
```

```
8}
```

С реализацией свойств по умолчанию в интерфейсах дело обстоит несколько сложнее, поскольку мы не можем определять в интерфейсах нестатические переменные, соответственно в свойствах интерфейса мы не можем манипулировать состоянием объекта. Тем не менее реализацию по умолчанию для свойств мы тоже можем определять:

```
1  interface IMovable
2  {
3      void Move()
4      {
5          Console.WriteLine("Walking");
6      }
7      // реализация свойства по умолчанию
8      // свойство только для чтения
9      int MaxSpeed { get { return 0; } }
10}
```

Стоит отметить, что если интерфейс имеет приватные методы и свойства (то есть с модификатором private), то они должны иметь реализацию по умолчанию. То же самое относится к любым статическим методам и свойствам (не обязательно приватным):

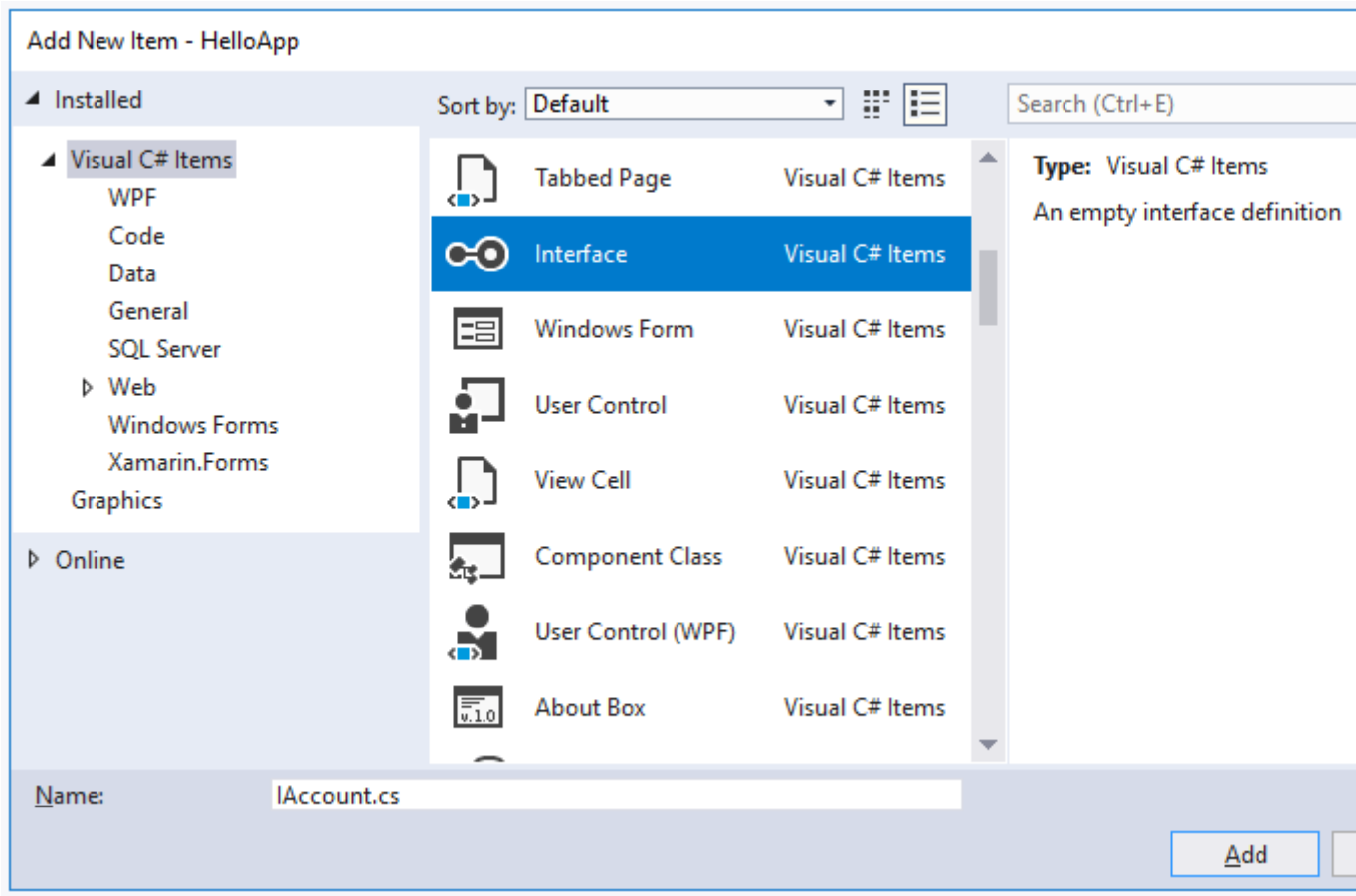
```
1  interface IMovable
2  {
3      public const int minSpeed = 0;     // минимальная скорость
4      private static int maxSpeed = 60;   // максимальная скорость
5          // находим время, за которое надо пройти расстояние distance со скоростью speed
6      static double GetTime(double distance, double speed) => distance / speed;
7      static int MaxSpeed
8      {
9          get { return maxSpeed; }
10         set
11         {
12             if (value > 0) maxSpeed = value;
13         }
14     }
15}
16class Program
17{
18     static void Main(string[] args)
19     {
20         Console.WriteLine(IMovable.MaxSpeed);
21         IMovable.MaxSpeed = 65;
22         Console.WriteLine(IMovable.MaxSpeed);
23         double time = IMovable.GetTime(100, 10);
24         Console.WriteLine(time);
25     }
26}
```

Модификаторы доступа интерфейсов

Как и классы, интерфейсы по умолчанию имеют уровень доступа **internal**, то есть такой интерфейс доступен только в рамках текущего проекта. Но с помощью модификатора public мы можем сделать интерфейс общедоступным:

```
1public interface IMovable
2{
3    void Move();
4}
```

Стоит отметить, что в Visual Studio есть специальный компонент для добавления нового интерфейса в отдельном файле. Для добавления интерфейса в проект можно нажать правой кнопкой мыши на проект и в появившемся контекстном меню выбрать **Add**-> **New Item...** и в диалоговом окне добавления нового компонента выбрать пункт **Interface**:



интерфейсы в C#.
При наследовании в C# в качестве родительского класса можно указать один и только один класс. Это не всегда бывает удобно. Для решения данной проблемы в C# вводится понятие интерфейса. В интерфейсе описываются только заголовки методов. Когда мы создаём класс, то можем указать, что данный класс реализует один интерфейс или сразу несколько.

Когда класс говорит, что он реализует какой-то интерфейс, класс гарантирует что внутри него (класса) реализованы все методы, которые описаны в интерфейсе. Интерфейс — это контракт, который необходимо исполнить классом.

Особенности интерфейсов:

1. Только заголовки методов.
2. Реализация методов отсутствует.
3. Нельзя создать экземпляр интерфейса, однако, можно создать переменную типа интерфейса, ведь это ссылочный тип данных. Такая переменная может указывать на объект класса, реализующего данный интерфейс.
4. Перед методом не нужно ставить "public", т.к. метод в интерфейс по умолчанию является "public" и "virtual".

Интерфейс может быть реализован также структурами.

Интерфейс создаётся с помощью ключевого слова "interface". Перед ключевым словом можно указать "public" или "internal" (по умолчанию, если ничего не указано):

```
public interface IFlyable
{

}
```

Что можно описать внутри интерфейсов:

1. Заголовки методов.
2. Можно описать свойства (properties), т.е. они не являются полями, это методы "get" и "set".
3. Индексатор (индексирующее property), т.к. это тоже метод.
4. Делегаты.

Как записать, что класс реализует интерфейс:

```
public class Animal: IFlyable
{

}
```

В данном случае мы указали, что класс "Animal" реализует интерфейс "IFlyable".

Если класс-родитель реализует какой-либо интерфейс, то считается, что и класс-наследник реализует данный интерфейс. В классе-наследнике реализацию можно переопределить. По рекомендации Microsoft перед названием всех интерфейсов нужно ставить букву "**I**". Технически, конечно, и без этого всё будет работать.

Интерфейс можно реализовать двумя способами:

1) Напрямую в классе, просто реализуя метод.

2) Прямая реализация метода интерфейса. В данном случае в классе создаётся метод, у которого не указывается уровень доступа, в качестве имени метода идёт имя интерфейса и имя метода. Это приведёт к тому, что не получится вызвать этот метод через ссылку самого объекта класса, можно вызывать только через ссылку типа интерфейса.

Интересный момент заключается в том, что можно сделать сразу двумя способами, причем по-разному реализовав, и это не будет перегрузкой, т.к. параметры будут одни и те же (и

одного и того же типа). Вызов данных методов зависит от типа ссылки, через которую они вызываются. В реальности обычно так не делают. Зачем так сделали мне, лично, непонятно.

Давайте на примере рассмотрим, как создавать и реализовывать интерфейсы:

```
// Что-то что летает
public interface IFlyable
{
  void fly();
}

// Животное, которое может летать
public class Bird : IFlyable
{
 // 1 способ реализации интерфейса
 public void fly()
 {
   Console.WriteLine("Птица полетела...");
 }
 // 2 способ реализации интерфейса
 void IFlyable.fly()
 {
   Console.WriteLine("Птица не полетела. Странно...");
 }
}
class Program
{
 static void Main(string[] args)
 {
   // Создадим переменную класса Bird
   Bird bird1 = new Bird();
   bird1.fly(); // выйдет "Птица полетела..."
   // Создадим переменную типа IFlyable
   IFlyable bird2 = new Bird();
   bird2.fly(); // выйдет "Птица не полетела. Странно..."
   // Если мы закоментируем метод "void IFlyable.fly()", то птица в обоих случаях полетит
   Console.ReadLine();
 }
}
```

Существует набор уже готовых интерфейсов, которые бывает полезно реализовать в собственных классах. Например, интерфейс "IComparable", который состоит из одного единственного метода "int CompareTo(Object obj)". Данный интерфейс даёт возможность сортировки объектов в коллекции. Для сортировки нужно уметь сравнивать 2 значения. Если это строки или числа, то здесь вопросов нет, а вот с объектами непонятно. Как сравнить, что один объект больше другого? А это вы сами должны описать в реализации этого интерфейса.