В C# разрешается вводить в программу информацию декларативного характера

в форме *атрибута,* с помощью которого определяются дополнительные сведения

(метаданные), связанные с классом, структурой, методом и т.д. Например, в программе можно указать атрибут, определяющий тип кнопки, которую должен отображать

конкретный класс. Атрибуты указываются в квадратных скобках перед тем элементом,

к которому они применяются. Следовательно, атрибут не является членом класса, но

обозначает дополнительную информацию, присоединяемую к элементу.

**Основы применения атрибутов**

Атрибут поддерживается классом, наследующим от класса System.Attribute.

Поэтому классы атрибутов должны быть подклассами класса Attribute. В классе

Attribute определены основные функциональные возможности, но далеко не все они

нужны для работы с атрибутами. В именах классов атрибутов принято употреблять

суффикс Attribute. Например, ErrorAttribute — это имя класса атрибута, описывающего

ошибку.

При объявлении класса атрибута перед его именем указывается атрибут

AttributeUsage. Этот встроенный атрибут обозначает типы элементов, к которым

может применяться объявляемый атрибут. Так, применение атрибута может ограничиваться

одними методами.

**Создание атрибута**

В классе атрибута определяются члены, поддерживающие атрибут. Классы атрибутов

зачастую оказываются довольно простыми и содержат небольшое количество

полей или свойств. Например, атрибут может определять примечание, описывающее

элемент, к которому присоединяется атрибут. Такой атрибут может принимать следующий

вид.

[AttributeUsage(AttributeTargets.All)]

public class RemarkAttribute : Attribute

{

string pri\_remark; // базовое поле свойства Remark

public RemarkAttribute(string comment)

{

pri\_remark = comment;

}

public string Remark

{

get

{

return pri\_remark;

}

}

}

Объявляемый атрибут получает имя RemarkAttribute. Его объявлению предшествует

встроенный атрибут AttributeUsage, указывающий на то, что атрибут

RemarkAttribute может применяться ко всем типам элементов. С помощью встроенного

атрибута AttributeUsage можно сузить перечень элементов, к которым может

присоединяться объявляемый атрибут. Подробнее о его возможностях речь пойдет

далее в этой главе.

Далее объявляется класс RemarkAttribute, наследующий от класса Attribute.

В классе RemarkAttribute определяется единственное закрытое поле pri\_remark,

поддерживающее одно открытое и доступное для чтения свойство Remark. Это свойство

содержит описание, связываемое с атрибутом. (Конечно, Remark можно было

бы объявить как автоматически реализуемое свойство с закрытым аксессором set, но

ради наглядности данного примера выбрано свойство, доступное только для чтения.)

В данном классе определен также один открытый конструктор, принимающий строковый

аргумент и присваивающий его свойству Remark. Этим пока что ограничиваются

функциональные возможности класса RemarkAttribute, готового к применению.

**Присоединение атрибута**

Как только класс атрибута будет определен, атрибут можно присоединить к элементу.

Атрибут указывается перед тем элементом, к которому он присоединяется, и

для этого его конструктор заключается в квадратные скобки. В качестве примера ниже

показано, как атрибут RemarkAttribute связывается с классом.

[RemarkAttribute("В этом классе используется атрибут.")]

class UseAttrib

{

// ...

}

В этом фрагменте кода конструируется атрибут RemarkAttribute, содержащий

комментарий "В этом классе используется атрибут." Данный атрибут затем

связывается с классом UseAttrib.

Присоединяя атрибут, совсем не обязательно указывать суффикс Attribute.

Например, приведенный выше класс может быть объявлен следующим образом.

[Remark("В этом классе используется атрибут.")]

class UseAttrib

{

// ...

}

В этом объявлении указывается только имя Remark. Такая сокращенная форма считается

вполне допустимой, но все же надежнее указывать полное имя присоединяемого

атрибута, чтобы избежать возможной путаницы и неоднозначности.

**Получение атрибутов объекта**

Как только атрибут будет присоединен к элементу, он может быть извлечен в других

частях программы. Для извлечения атрибута обычно используется один из двух

методов. Первый метод, GetCustomAttributes(), определяется в классе MemberInfо

и наследуется классом Туре. Он извлекает список всех атрибутов, присоединенных к

элементу. Ниже приведена одна из его форм.

object[] GetCustomAttributes(bool наследование)

Если *наследование* имеет логическое значение true, то в список включаются

атрибуты всех базовых классов, наследуемых по иерархической цепочке. В противном

случае атрибуты извлекаются только из тех классов, которые определяются указанным

типом.

Второй метод, GetCustomAttribute(), определяется в классе Attribute. Ниже

приведена одна из его форм:

static Attribute GetCustomAttribute(MemberInfo элемент, Type тип\_атрибута)

где *элемент* обозначает объект класса MemberInfo, описывающий тот элемент, для которого

создаются атрибуты, тогда как *тип\_атрибута* — требуемый атрибут. Данный

метод используется в том случае, если имя получаемого атрибута известно заранее, что

зачастую и бывает. Так, если в классе UseAttrib имеется атрибут RemarkAttribute,

то для получения ссылки на этот атрибут можно воспользоваться следующей последовательностью кода.

// Получить экземпляр объекта класса MemberInfо, связанного

// с классом, содержащим атрибут RemarkAttribute.

Type t = typeof(UseAttrib);

// Извлечь атрибут RemarkAttribute.

Type tRemAtt = typeof(RemarkAttribute);

RemarkAttribute ra = (RemarkAttribute)

Attribute.GetCustomAttribute(t, tRemAtt);

Эта последовательность кода оказывается вполне работоспособной, поскольку класс

MemberInfo является базовым для класса Туре. Следовательно, t — это экземпляр

объекта класса MemberInfo.

Имея ссылку на атрибут, можно получить доступ к его членам. Благодаря этому информация

об атрибуте становится доступной для программы, использующей элемент,

к которому присоединен атрибут. Например, в следующей строке кода выводится содержимое

свойства Remark.

Console.WriteLine(rа.Remark);

Ниже приведена программа, в которой все изложенные выше особенности применения

атрибутов демонстрируются на примере атрибута RemarkAttribute.

(***glava17\_9***)

[AttributeUsage(AttributeTargets.All)]

public class RemarkAttribute : Attribute

{

string pri\_remark;

public RemarkAttribute(string comment)

{

pri\_remark = comment;

}

public string Remark

{

get

{

return pri\_remark;

}

}

}

[RemarkAttribute("This class use Attribute.")]

class UseAttrib

{

}

class AttribDemo

{

static void Main()

{

Type t = typeof(UseAttrib);

Console.Write("Attributes in class " + t.Name + ": ");

object[] attribs = t.GetCustomAttributes(false);

foreach (object o in attribs)

Console.WriteLine(o);

Console.WriteLine("Remark: ");

//execute attribute Remark

Type tRemAtt = typeof(RemarkAttribute);

RemarkAttribute ra = (RemarkAttribute)

Attribute.GetCustomAttribute(t, tRemAtt);

Console.WriteLine(ra.Remark);

}

}

**Сравнение позиционных и именованных параметров**

В предыдущем примере для инициализации атрибута RemarkAttribute его

конструктору была передана символьная строка с помощью обычного синтаксиса

конструктора. В этом случае параметр comment конструктора RemarkAttribute()

называется *позиционным.* Этот термин отражает тот факт, что аргумент связан с параметром

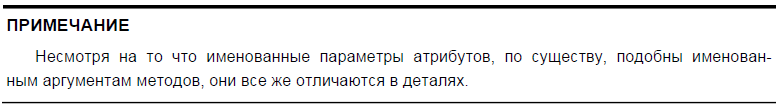
по его позиции в списке аргументов. Следовательно, первый аргумент передается

первому параметру, второй аргумент — второму параметру и т.д.

Но для атрибута доступны также *именованные параметры,* которым можно присваивать

первоначальные значения по их именам. В этом случае значение имеет имя,

а не позиция параметра.



Именованный параметр поддерживается открытым полем или свойством, которое

должно быть нестатическим и доступным только для записи. Любое поле или свойство

подобного рода может автоматически использоваться в качестве именованного

параметра. Значение присваивается именованному параметру с помощью соответствующего

оператора, расположенного в списке аргументов при вызове конструктора

атрибута. Ниже приведена общая форма объявления атрибута, включая именованные

параметры.

[attrib(список\_позиционных\_параметров,

именованный\_параметр\_1 = значение,

именованный\_параметр\_2 = значение, ...)]

Первыми указываются позиционные параметры, если они существуют. Далее следуют

именованные параметры с присваиваемыми значениями. Порядок следования

именованных параметров особого значения не имеет. Именованным параметрам не

обязательно присваивать значение, и в этом случае используется значение, устанавливаемое

по умолчанию.

Применение именованного параметра лучше всего показать на конкретном примере.

Ниже приведен вариант класса RemarkAttribute, в который добавлено поле

Supplement, предназначенное для хранения дополнительного примечания.

public class RemarkAttribute : Attribute

{

string pri remark; // базовое поле свойства Remark

// Это поле можно использовать в качестве именованного параметра.

public string Supplement;

public RemarkAttribute(string comment)

{

pri\_remark = comment;

Supplement = "Отсутствует";

}

public string Remark

{

get

{

return pri\_remark;

}

}

}

Как видите, поле Supplement инициализируется в конструкторе символьной строкой

"Отсутствует". Другого способа присвоить ему первоначальное значение в конструкторе

не существует. Но поскольку поле Supplement является открытым в классе

RemarkAttribute, его можно использовать в качестве именованного параметра, как

показано ниже.

[RemarkAttribute("В этом классе используется атрибут.",

Supplement = "Это дополнительная информация.")]

class UseAttrib

{

// ...

}

Обратите особое внимание на вызов конструктора класса RemarkAttribute. В этом

конструкторе первым, как и прежде, указывается позиционный параметр, а за ним

через запятую следует именованный параметр Supplement, которому присваивается

конкретное значение. И наконец, закрывающая скобка, ), завершает вызов конструктора.

Таким образом, именованный параметр инициализируется в вызове конструктора.

Этот синтаксис можно обобщить: позиционные параметры должны указываться в том

порядке, в каком они определены в конструкторе, а именованные параметры — в произвольном

порядке и вместе с присваиваемыми им значениями.

Ниже приведена программа, в которой демонстрируется применение поля

Supplement в качестве именованного параметра атрибута.

(***glava17\_10***)

[AttributeUsage(AttributeTargets.All)]

public class RemarkAttribute : Attribute

{

string pri\_remark; //base field property Remark

public string Supplement; //named parameter

public RemarkAttribute(string comment)

{

pri\_remark = comment;

Supplement = "Missed";

}

public string Remark

{

get

{

return pri\_remark;

}

}

}

[RemarkAttribute("This class has Attribute.", Supplement = "This is additional info.")]

class UseAttrib

{

//....

}

class NamedParamDemo

{

static void Main()

{

Type t = typeof(UseAttrib);

Console.Write("Attribute in class " + t.Name +

": ");

object[] attribs = t.GetCustomAttributes(false);

foreach (object o in attribs)

{

Console.WriteLine(o);

}

//get attribute

Type tRemAtt = typeof(RemarkAttribute);

RemarkAttribute ra = (RemarkAttribute)

Attribute.GetCustomAttribute(t, tRemAtt);

Console.Write("Remark: ");

Console.WriteLine(ra.Remark);

Console.Write("Additional: ");

Console.WriteLine(ra.Supplement);

}

}

Прежде чем перейти к следующему вопросу, следует особо подчеркнуть, что поле

***pri\_remark*** ***нельзя***использовать в ***качестве именованного параметра***, поскольку оно

закрыто в классе RemarkAttribute. Свойство Remark также ***нельзя***использовать в качестве

именованного параметра, потому что оно доступно только для чтения. Напомним,

что в качестве именованных параметров могут служить только открытые поля и

свойства.

Открытое и доступное только для чтения свойство может использоваться в качестве

именованного параметра таким же образом, как и открытое поле. В качестве примера

ниже показано, как автоматически реализуемое свойство Priority типа int вводится

в класс RemarkAttribute.

(***glava17\_11***)

[AttributeUsage(AttributeTargets.All)]

public class RemarkAttribute : Attribute

{

string pri\_remark; //base field property Remark

public string Supplement; //named parameter

public RemarkAttribute(string comment)

{

pri\_remark = comment;

Supplement = "Missed";

Priority = 1;

}

public string Remark

{

get

{

return pri\_remark;

}

}

//use as property named parameter

public int Priority { get; set; }

}

[RemarkAttribute("This class has Attribute.", Supplement = "This is additional info.",

Priority = 10)]

class UseAttrib

{

//....

}

class NamedParamDemo

{

static void Main()

{

Type t = typeof(UseAttrib);

Console.Write("Attribute in class " + t.Name +

": ");

object[] attribs = t.GetCustomAttributes(false);

foreach (object o in attribs)

{

Console.WriteLine(o);

}

//get attribute

Type tRemAtt = typeof(RemarkAttribute);

RemarkAttribute ra = (RemarkAttribute)

Attribute.GetCustomAttribute(t, tRemAtt);

Console.Write("Remark: ");

Console.WriteLine(ra.Remark);

Console.Write("Additional: ");

Console.WriteLine(ra.Supplement);

Console.WriteLine("Priority: " + ra.Priority);

}

}

Именованные параметры атрибутов Supplement и Priority *не* обязательно указывать

в каком-то определенном порядке. Порядок их указания можно свободно изменить,

не меняя сами атрибуты.

И последнее замечание: тип параметра атрибута (как позиционного, так и именованного)

должен быть одним из встроенных простых типов, object, Туре, перечислением

или одномерным массивом одного из этих типов.

**Встроенные атрибуты**

В C# предусмотрено несколько встроенных атрибутов, но три из них имеют особое

значение, поскольку они применяются в самых разных ситуациях. Это атрибуты

AttributeUsage, Conditional и Obsolete, рассматриваемые далее по порядку.

**Атрибут AttributeUsage**

Как упоминалось ранее, атрибут AttributeUsage определяет типы элементов, к

которым может быть применен объявляемый атрибут. AttributeUsage — это, по

существу, еще одно наименование класса System.AttributeUsageAttribute. У него

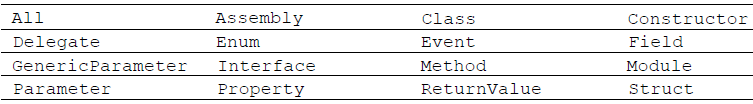
имеется следующий конструктор:

AttributeUsage(AttributeTargets validOn)

где *validOn* обозначает один или несколько элементов, к которым может быть применен

объявляемый атрибут, тогда как AttributeTargets — перечисление, в котором

определяются приведенные ниже значения.



Два этих значения или более можно объединить с помощью логической операции

ИЛИ. Например, для указания атрибута, применяемого только к полям и свойствам,

используются следующие значения.

AttributeTargets.Field | AttributeTargets.Property

В классе атрибута AttributeUsage поддерживаются два именованных параметра.

Первым из них является параметр AllowMultiple, принимающий логическое значение.

Если это значение истинно, то атрибут может быть применен к одному и тому же

элементу неоднократно. Второй именованный параметр, Inherited, также принимает

логическое значение. Если это значение истинно, то атрибут наследуется производными

классами, а иначе он не наследуется. По умолчанию параметр AllowMultiple принимает

ложное значение (false), а параметр Inherited — истинное значение (true).

В классе атрибута AttributeUsage определяется также доступное только для чтения

свойство ValidOn. Оно возвращает значение типа AttributeTargets, определяющее

типы элементов, к которым можно применять объявляемый атрибут. По умолчанию

используется значение AttributeTargets.All.

**Атрибут Conditional**

Атрибут Conditional представляет, вероятно, наибольший интерес среди всех

встроенных атрибутов. Ведь он позволяет создавать *условные методы,* которые вызываются

только в том случае, если с помощью директивы #define определен конкретный

идентификатор, а иначе метод пропускается. Следовательно, условный метод служит

альтернативой условной компиляции по директиве #if.

Conditional — это, по существу, еще одно наименование класса System.

Diagnostics.ConditionalAttribute. Для применения атрибута Conditional в исходный

код программы следует включить пространство имен System.Diagnostics.

Рассмотрим применение данного атрибута на следующем примере программы.

(***glava17\_12***)

#define TRIAL

using System;

using System.Diagnostics;

class Test

{

[Conditional("TRIAL")]

void Trial()

{

Console.WriteLine("Trial version, not for upload.");

}

[Conditional("RELEASE")]

void Release()

{

Console.WriteLine("Ended working version.");

}

static void Main()

{

Test t = new Test();

t.Trial(); //will call if defined identificator TRIAL

t.Release(); //will call if defined indentificator RELEASE

}

}

Рассмотрим эту программу подробнее, чтобы стал понятнее результат ее выполнения.

Прежде всего обратите внимание на то, что в этой программе определяется

идентификатор TRIAL. Затем обратите внимание на определение методов Trial() и

Release(). Каждому из них предшествует атрибут Conditional, общая форма которого

приведена ниже:

[Conditional(идентификатор)]

где *идентификатор* обозначает конкретный идентификатор, определяющий условие

выполнение метода. Данный атрибут может применяться только к методам. Если

идентификатор определен, то метод выполняется, когда он вызывается. Если же идентификатор

не определен, то метод не выполняется.

Оба метода, Trial() и Release(), вызываются в методе Main(). Но поскольку

определен один лишь идентификатор TRIAL, то выполняется только метод Trial(),

тогда как метод Release() игнорируется. Если же определить идентификатор

RELEASE, то метод Release() будет также выполняться. А если удалить определение

идентификатора TRIAL, то метод Trial() выполняться не будет.

Атрибут Conditional можно также применить в классе атрибута, т.е. в классе,

наследующем от класса Attribute. Так, если идентификатор определен, то атрибут

применяется, когда он встречается в ходе компиляции. В противном случае он не применяется.

На условные методы накладывается ряд ограничений. Во-первых, они должны возвращать

значение типа void, а по существу, ничего не возвращать. Во-вторых, они

должны быть членами класса или структуры, а не интерфейса. И в-третьих, они не

могут предшествовать ключевому слову override.

**Атрибут Obsolete**

Атрибут Obsolete (сокращенное наименование класса System.ObsoleteAttribute)

позволяет пометить элемент программы как устаревший. Ниже приведена общая форма

этого атрибута:

[Obsolete("сообщение")]

где *сообщение* выводится при компилировании элемента программы, помеченного

как устаревший. Ниже приведен краткий пример применения данного атрибута.

(***glava17\_13***)

using System;

class Test

{

[Obsolete("Better to use Method MyMeth2.")]

public static int MyMeth(int a, int b)

{

return a / b;

}

//new variant MyMeth

public static int MyMeth2(int a, int b)

{

return b == 0 ? 0 : a / b;

}

static void Main()

{

//for this code will output warning

Console.WriteLine("4 / 3 = " + Test.MyMeth(4, 3));

//not for this

Console.WriteLine("4 / 3 = " + Test.MyMeth2(4, 3));

}

}

Когда по ходу компиляции программы в методе Main() встречается вызов метода

MyMeth(), формируется предупреждение, уведомляющее пользователя о том, что ему

лучше воспользоваться методом MyMeth2().

Ниже приведена вторая форма атрибута Obsolete:

[Obsolete("сообщение", ошибка)]

где *ошибка* обозначает логическое значение. Если это значение истинно (true), то при

использовании устаревшего элемента формируется сообщение об ошибке компиляции

вместо предупреждения. Эта форма отличается тем, что программа, содержащая

подобную ошибку, не будет скомпилирована в исполняемом виде.