**Özniteliklerin (Attributes) Kullanımı**

Öznitelikler modern programlama dillerine gittikçe daha fazla girmeye başlamıştır. C#’ta bir bildirimin önüne getirilen köşeli parantezli ifadeler özenitelik belirtir. Öznitelikler C#’ta şu bildirimlerin önüne getirilebilir:

- Sınıflar

- Yapılar

- Arayüzler

- Enum’lar

- Metotlar

- Veri elemanları

- Metot parametreleri

- Metotların geri dönüş değerleri

- Delegeler

- Property’ler

Ayrıca öznitelikler assembly dosyalarının bizzat kendisine de iliştirilebilmektedir.

Köşeli parantezler içerisinde bir sınıf ismi bulunmak zorundadır. Köşeli parantezler içerisinde bulundurulacak sınıflar öznitelik sınıfları denir. Bütün öznitelik sınıfları System.Attribute sınıfı türünden ya da bu sınıftan türetilmiş sınıflar türünden olmak zorundadır. .NET kütüphanesinde yalnızca System.Attribute sınıfı yoktur. Bu sınıftan türetilmiş pek çok öznitelik sınıfı da vardır. Tabi biz de System.Attribute sınıfından türetme yaparak kendi öznitelik sınıflarımızı oluşturabiliriz.

Köşeli parantezlerin içerisinde öznitelik sınıflarının bir başlangıç metodunun çağrılmasına ilişkin bir ifadenin bulunması gerekir. Örneğin:

using System;

namespace CSD

{

class MyAttribute : Attribute {

private string m\_msg;

public MyAttribute(string msg)

{

m\_msg = msg;

}

}

[MyAttribute("Bu bir denemedir")]

class Sample {

}

public class App

{

public static void Main()

{

}

}

}

Burada MyAttribute sınıfının string parametreli bir başlangıç metodu vardır. Sample sınıfına bu öznitelik iliştirilirken parantezler içerisinde sanki bu başlangıç metodu çağrılacakmış gibi ilgili türden argüman girilmiştir.

C# derleyicisi derleme sırasında bir öznitelik kullanımıyla karşılaştığında belirtilen başlangıç metodunu kullanarak o öznitelik sınıfı türünden bir nesne yaratır. O nesnenin veri elemanlarını assembly dosyası içerisine gömer. Böylece bir bildirim ile bir sınıf nesnesi derleyici tarafından eşleştirilmiş olur. Programcı isterse programın çalışma zamanı sırasında derleyicinin derleme aşamasında yaratıp assembly’ye yerleştirdiği bu nesneyi yeniden elde edebilir.

Bir öznitelik nesnesi programın çalışma zamanı sırasında şöyle elde edilebilir:

1) Reflection yöntemi ile elde edilebilir. Bu konu ileride ele alınacaktır.

2) Type sınfının GetCustomAttributes metotları ile elde edilebilir. Type sınıfının GetCustomAttributes metodu şu şekildedir.

public abstract Object[] GetCustomAttributes(bool inherit)

Buradaki parametre o sınıfın taban sınıflarındaki özniteliklerin de elde edilip edilmeyeceğini belirtir. Örneğin:

[YourAttribute(123)]

[MyAttribute("This is a test")]

class Sample

{

//...

}

class MyAttribute : Attribute

{

private string m\_msg;

public MyAttribute(string msg)

{

m\_msg = msg;

}

public string Msg { get { return m\_msg; } }

//...

}

class YourAttribute : Attribute

{

private int m\_no;

public YourAttribute(int no)

{

m\_no = no;

}

public int No { get { return m\_no; } }

//...

}

Burada Sample sınıfına MyAttribute ve YourAttribute isimli iki öznitelik nesnesi iliştirilmiştir. Derleyici bu nesneleri yaratıp onların bilgilerini assembly dosyasına yerleştirir. Biz de istedğimiz zaman onu oradan aşağıdaki gibi alıp kullanabiliriz:

Type sampleType = typeof(Sample);

object[] objs;

objs = sampleType.GetCustomAttributes(false);

foreach (object o in objs)

{

if (o is MyAttribute)

{

MyAttribute ma = (MyAttribute)o;

//...

}

else if (o is YourAttribute)

{

YourAttribute ya = (YourAttribute)o;

//...

}

}

Type sınıfının aşağıdaki overload edilmiş GetCustomAttributes metodu yalnıznca istediğimiz bir türün özniteliklerini bize verir:

**public** **abstract** [Object](ms-help://MS.MSDNQTR.v90.en/fxref_mscorlib/html/ee2c26d9-17cc-ab19-8a9c-6fca33a3c7ad.htm)[] GetCustomAttributes(

[Type](ms-help://MS.MSDNQTR.v90.en/fxref_mscorlib/html/abee003d-eb9f-f380-7902-6af6cb34a622.htm) *attributeType*,

[bool](ms-help://MS.MSDNQTR.v90.en/fxref_mscorlib/html/ff35b1f1-386c-370b-2c36-a48e7dcbc147.htm) *inherit*

)

Örneğin:

objs = sampleType.GetCustomAttributes(typeof(MyAttribute), false);

Öznitelik sınıflarının sonunun XXXAttribute biçiminde bitirilmesi gelenekseldir. Eğer bir öznitelik sınıfı XXXAttribute biçiminde isimlendirilmişse biz onu köşeli aparantezler içerisinde yalnızca XXX ismiyle kullanabiliriz. Eğer hem XXXAttribute hem de XXXAttributeAttirbute isimli öznitelik sınıfları varsa biz [XXXAttribute] biçiminde belirleme yaptığımızda XXXAttribute sınıfı anlaşılır. (Eğer XXXAttribute isimli bir sınıf olmasaydı bu durumda XXXAttributeAttribute sınıfı anlaşılırdı) Benzer biçimde hem XXX isimli hem de XXXAttribute isimli iki sınıf olsa biz köşeli parantez içerisinde XXX yazarsak XXX sınıf anlaşılır.

Eğer öznitelik nesnesi ilgili sınıfın default başlangıç metodu ile yaratılacaksa bu durumda parantezlerin kullanılmasına gerek yoktur. Örneğin aşağıdaki bildirimlerin hepsi aynı anlamdadır.

[MyAttribute]

class Sample

{

//...

}

ile

[MyAttribute()]

class Sample

{

//...

}

ile,

[My]

class Sample

{

//...

}

ile

[My()]

class Sample

{

//...

}

Aynı bildirime birden fazla öznitelik iliştirilirken bu işlem farklı köşeli parantezlerle de tek bir köşeli parantezle de yapılabilir. Örneğin:

[YourAttribute(123)]

[MyAttribute("This is a test")]

class Sample

{

//...

}

ile,

[YourAttribute(123), MyAttribute("This is a test")]

class Sample

{

//...

}

aynı anlamdadır. Özniteliklerin belirtilme sırasının hiçbir önemi yoktur.

**Özniteliklere Neden Gereksinim Duyulmaktadır?**

Öznitelikler bir bildirime iliştirilir. Sonra herhangi bir kaynak tarafından bu öznitelikler elde edilip belli bir davranış için ipucu olarak kullanılabilir. Bazı öznitelikleri bazı framework’ler, bazı kütüphaneler (hatta bizzat C# derleyicisinin kendisi bile) kontrol edip davranışını buna göre yönlendirebilmektedir. Burada birkaç örnekle kullanım konusunda fikir vermeye çalışalım.

ObsoleteAttribute isimli öznitelik sınıfı bizzat C# derleyicisinin kendisi tarafından kullanılmaktadır. C# derleyicisi bir bildirime ObsoleteAttribute özniteliği iliştirildiğini gördüğünde “ilgili sınıf, metot ya da elemanın artık daha iyi bir versiyonunun olduğuna, bunun artık kullanılmaması gerektiğine ilişkin” bir mesaj verir. Örneğin:

class Test

{

[Obsolete]

public static void Foo()

{

//...

}

}

Yine örneğin ConditinalAttribute isimli öznitelik sınıfı string türünden bir başlangıç metoduna sahiptir. Bu öznitelik sınıfı bir metoda iliştirilebilir. Bu durumda bu metot çağrıldığında eğer programın tepesinde öznitelik nesnesinde belirtilen sembolik sabit #define edilmişse çağrı koda dahil edilir. Aksi taktirde çağrı sanki hiç yapılmamış gibi derleyici tarafından ele alınır. Örneğin:

#define TEST

using System;

using System.Diagnostics;

namespace CSD

{

class Sample {

[Conditional("TEST")]

public void Foo()

{

Console.WriteLine("Foo");

}

}

public class App

{

public static void Main()

{

Sample s = new Sample();

s.Foo();

}

}

}

Burada biz programın çeşitli yerlerinde Test.Foo() çağrısını yaptığımızı düşünelim. Eğer programın tepesinde

#define TEST

direktifi varsa bu çağrı koda dahil edilmektedir. Aksi taktirde derleyici tarafından bu çağrılar elemine edilir. #define direktifini kaynak kodun tepesine yerleştirmek yerine proje seçeneklerinde Build/Conditional compilation symbols kısmına da girilebilir.

Bu öznitelik sayesinde programların Debug ve Release versiyonları oluşturulabilmektedir.

Visual Studio IDE’sinde bir proje için baştan iki konfigürasyon yaratılmaktadır: Debug ve Release. Tabii programcı isterse kendi konfigürasyonlarını da yaratabilir. Projenin Debug versiyonunda “DEBUG” isimli sembolik sabir #define direktifi ile bildirilmiş kabul edilir. Fakat Release’de bu sembolik sabit bildirilmiş kabul edilmez. O halde biz bir metodun önüne Conditional(“DEBUG”) özniteliğini iliştirirsek bu metot çağrıları yalnızca DEBUG versiyonunda koda dahil edilir. Projenin Release versiyonunda koda dahil edilmez.

İşte biz projeyi geliştirirken hataları yakalayabilmek için pek çok kontrolü projenin debug versiyonunda yapabiliriz. Bu kontrolleri yapan metotlar Conditional(“DEBUG”)] ile özniteliklendirildiği için projenin release versiyonunda tamamen koddan çıkartılırlar. Aslında .NET kütüphanelerinin de bu biçimde Debug ve Release versiyonları vardır. Biz proje ayarlarını Debug versiyona çektiğimizde o kütüphaneleri kullanmış oluruz. O kütüphanelrin içerisnde pek çok kontrol yapılmıştır. Fakat biz konfigürasyonu Release’e çektiğimizde bu sefer kütüphanenin kontrolsüz ve hızlı çalışan kütüphaneleri devreye girer. Böylece biz projemizi debug versiyonunda geliştiririz. Bu versiyon bizim bug’larımızı ekstra kontrollerle daha iyi görmektedir. Fakat programımızı satarken Release versiyona geçerek onu son kez derlemeliyiz. Müşteriye teslim edeceğimiz versiyon release versiyon olmalıdır.

Bazı öznitelikler bazı framework’ler tarafından kullanılmaktadır. Örneğin bir framework kodumuz içerisindeki bir sınıfı özel bir amaçla kullanmak isteyebilir. Bu tespit edebilmek için bizim o sınıfı o öznitelikle niteliklendirmemiz gerekir. Örneğin seri hale getirme mekanizmasında kütüphane seri hale getirilecek sınıfları tespit edip ona bazı operarasyonlar uygular. Bizim de o sınıfları SerializableAttrtibute ile özniteliklendirmemiz gerekir.

**Öznitelik Sınıflarının Konumsal ve İsimli Argümanları**

Öznitelik sınıflarının kesinlikle girmek zorunda olduğumuz argümanlarına konumsal (positional) argümanlar, girip girmemekte serbest olduğumuz argümanlara da isimli (named) argümanlar denilmektedir. Öznitelik sınıflarının başlangıç metot parametrelerine karşı gelen argümanları konumsal argümanlardır. Bunlar o sırada köşeli parantez içerisinde belirtilmek zorundadır. Ancak öznitelik sınıflarının public veri elemanları ve write özelliği olan property’leri isimli argümanlardır. Onlar belirtilse de belirtilmese de olur. İsimli argümanlar “İsim = değer” biçiminde aralarına virgül konularak belirtilir. Önce konumsal argümanların sonra isimli argümanların girilmesi gerekir. Ancak isimli argümanların sırasının bir önemi yoktur. Örneğin:

[Test("This is a test", Val = 123)]

class Sometghing

{

//...

}

class TestAttribute : Attribute

{

private string m\_msg;

private int m\_val;

public TestAttribute(string msg)

{

m\_msg = msg;

}

public int Val

{

get { return m\_val; }

set { m\_val = value; }

}

}

Burada TestAttribute sınıfının string parametreli başlangıç metodu vardır. Bu başlangıç metodunun string parametresi mutlaka belirtilmek zorundadır. Bu nedenle ona konumsal argüman denir. Halbuki Val property’si isimli argümandır. Biz onu belirtebiliriz ya da belirtmeyebiliriz. Tabii biz isimli argümanları hiç belirtmezsek ilgili öznitelik sınıfı onlar için bazı default değerler almaktadır:

using System;

using System.Diagnostics;

namespace CSD

{

class MyAttribute : Attribute {

public string Message { get; private set; }

public int Val { get; set; }

public MyAttribute(string msg)

{

Val = 30;

Message = msg;

}

}

[My("Test", Val = 20)]

class Sample {

public void Foo()

{

Console.WriteLine("Foo");

}

}

public class App

{

public static void Main()

{

Type t = typeof(Sample);

MyAttribute ma = (MyAttribute)t.GetCustomAttributes(false)[0];

Console.WriteLine(ma.Val);

}

}

}

**Özniteliklere İlişkin Diğer Özellikler**

Bazı öznitelik sınıflarının özel olarak yalnızca belirli bildirimlerin önüne getirilmesi istenebilir. Bunu sağlamak için öznitelik sınıfının kendisini AttrbiuteUsageAttribute sınıfıyla özniteliklendirmek gerekir. AttributesUsageAttribute sınıfının başlangıç metodunun parametresi (yani konumsal parametresi) AttributeTargets isimli bir enum türündendir. Bu enum türünün elemanları ‘|’ operatörüyle birleştirilebilir. Böylece birden fazla eleman belirtilmiş olur. Bu enum türünün elemanları şunlardır:

Assembly

Module

Class

Struct

Enum

Constructor

Method

Property

Field

Event

Interface

Parameter

Delegate

ReturnValue

GenericParameter

All

Örneğin:

[AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct)]

class TestAttribute : Attribute

{

private string m\_msg;

private int m\_val;

public TestAttribute(string msg)

{

m\_msg = msg;

}

public int Val

{

get { return m\_val; }

set { m\_val = value; }

}

}

Burada yalnızca sınıf ve yapılar bu TestAttribute sınıfı ile özniteliklendirebilir. Ancak örneğin metotları, parametre değişkenlerini, arayüzleri bu sınıfla özniteliklendiremeyiz.

Normal olarak bir bildirim aynı öznitelik sınıfyla birden fazla kez özniteliklendirilemez. Örneğin:

[Test("This is a test", Val = 123)]

[Test("This is another test")] // error!

class Something

{

//...

}

Eğer bunun yapılması isteniyorsa AttributeUsageAttribute sınıfının AllowMultiple “isimli parametresi” true yapılmalıdır. Örneğin:

[AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct, AllowMultiple = true)]

class TestAttribute : Attribute

{

//...

}

Örneğin:

using System;

using System.Diagnostics;

namespace CSD

{

[AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct, AllowMultiple=true)]

class MyAttribute : Attribute {

public string Message { get; private set; }

public int Val { get; set; }

public MyAttribute(string msg)

{

Val = 30;

Message = msg;

}

public override string ToString()

{

return string.Format("Message:{0} Val:{1}", Message, Val);

}

}

[My("Test", Val = 20)]

[My("Merhaba", Val = 300)]

class Sample {

public void Foo()

{

Console.WriteLine("Foo");

}

}

public class App

{

public static void Main()

{

object[] objs = typeof(Sample).GetCustomAttributes(typeof(MyAttribute), false);

foreach (MyAttribute ma in objs) {

Console.WriteLine(ma);

//...

}

}

}

}