8

Attribuutit

Useimmat ohjelmointikielet on suunniteltu tavoitteena määrätty joukko ominaisuuksia. Kun esimerkiksi ryhdyt tekemään kääntäjää, mietit millainen uudella kielellä tehty sovellus on rakenteeltaan, miten koodissa kutsutaan toista koodia, miten toiminnot paketoidaan ja monia muita asioita, jotka tekevät kielestä tuottavan työkalun ohjelmien kehittämiseen. Useimmista kielen ominaisuuksista tulee staattisia. Esimerkiksi C#:ssa määrittelet luokan kirjoittamalla avainsanan *class* sen nimen eteen. Periytymisen ilmoitat kirjoittamalla puolipisteen luokan nimen perään ja sen perään kantaluokan nimen. Nämä ovat esimerkkejä päätöksistä, jotka kielen suunnittelijat ovat tehneet ja joita ei sen jälkeen enää voi muuttaa.

Kääntäjiä tekevät ihmiset ovat pahuksen hyviä siinä. Mutta edes he eivät voi ennakoida kaikkea alan kehitystä eikä sitä, miten se vaikuttaa tapaan, jolla ohjelmoijat tulevaisuudessa haluavat ilmaista omat tyyppinsä. Esimerkiksi miten luot yhteyden C++-luokan ja tuon luokan dokumentin sisältävän URL:n välille? Tai miten liität määrätyt C++-luokan jäsenet XML-kenttiin yrityksesi uudessa business-to-business-sovelluksessa? Koska C++ on suunniteltu vuosia ennen Internetin ja XML:n tapaisten protokollien yleistymistä, ei sen avulla ole helppoa tehdä kumpaakaan edellä kuvattua tehtävää.

Tähän asti ratkaisu tällaisiin ongelmiin on ollut tallentaa lisätiedot erillisiin tiedostoihin (DEF, IDL jne.), jotka liitetään löysästi kyseiseen tyyppiin tai jäseneen. Koska kääntäjällä ei ole mitään tietoa erillisistä tiedostoista tai koodilla generoitua yhteyttä luokan ja tiedoston välillä, sanotaan tällaista ratkaisua usein yhteydettömäksi ratkaisuksi. Suurin ongelma tästä on se, että luokka ei ole enää "itsekuvaava", eli käyttäjä ei voi enää selvittää luokan määrittelystä kaikkia siihen liittyviä asioita. Itsekuvaavan komponentin yksi etu on se, että kääntäjä ja ajonaikainen ympäristö voivat varmistaa, että komponenttiin liitettyjä

sääntöjä noudatetaan. Lisäksi itsekuvaava komponentti on helpompi ylläpitää, koska ohjelmoija näkee kaiken komponenttiin liittyvän informaation samasta paikasta.

Tämä on ollut vuosikymmeniä tilanne kääntäjien kehityksessä. Kielen tekijät yrittävät määritellä, mitä sinun pitää kielellä tehdä, suunnittelevat kääntäjän noiden tarpeiden mukaan ja nuo ominaisuudet sinulla on käytössäsi, kunnes toinen kääntäjä ilmestyy. Näin siis tähän asti, kunnes C# tarjoaa nyt erilaisen mahdollisuuden, joka on seurausta attribuutit-nimisen piirteen esittelystä.

Johdanto attribuutteihin

Se, mitä attribuutit mahdollistavat, ei ole suorastaan uraauurtavaa. Ne tarjoavat yleisen tavan liittää informaatiota määrittelemääsi C#-tyyppiin. Voit käyttää attribuutteja suunnittelun aikaisen informaation määrittelemiseen (kuten dokumentointitietoja), suorituksen aikaiseen informaatioon (kuten kentän nimi tietokannan taulussa), tai jopa suorituksen aikaiseen käyttäytymisominaisuuksiin (kuten onko määrätty jäsen mukana tapahtumassa, eli osallistuuko se transaktioon). Mahdollisuudet ovat loputtomat, ja se on on juuri se uusi suuri asia. Koska voit luoda attribuutin, joka perustuu mihin tahansa haluamaasi informaatioon, on attribuuttien määrittämiseen luotu standardi, kuten myös jäseneen tai tyyppiin liitetyn attribuutin arvon kyselemiseen suorituksen aikana.

Esimerkki kuvaa paremmin tätä tehokasta ominaisuutta. Sanotaan, että sinulla on sovellus, joka tallentaa osan tiedoistaan Windowsin Rekisteriin. Yksi suunnittelupäätös on se, minne tallennetaan Rekisterin avaintieto. Useimmissa kehitysympäristöissä se normaalisti tallennettaisiin resurssitiedostoon tai vakioon tai se jopa kovakoodattaisiin Rekisterin APIn kutsuihin. Kuitenkin, kuten edellä mainitsin, tässä meillä on tilanne, jossa oleellinen osa luokkaa tallennetaan erilleen luokan muusta määrittelystä. Attribuuttien avulla voimme liittää tämän tiedon luokan jäseneen, jolloin meillä on täysin itsensä kuvaava komponentti. Tässä esimerkki, miltä tämän toteuttaminen näyttää, olettaen, että meillä on olemassa muualla määritelty *RegistryKey*-attribuutti:

```
class MyClass
{
    [RegistryKey(HKEY_CURRENT_USER, “foo”)]
    public int Foo;
}
```

Määritellyn attribuutin liittäminen C#-tyyppiin tai -jäseneen onnistuu yksinkertaisesti kirjoittamalla attribuutin tiedot hakasulkeisiin ennen kohdetyyppiä tai -jäsentä. Tässä olemme liittäneet *RegistryKey*-nimisen attribuutin *MyClass.Foo*-kenttään. Suorituksen aikana, kuten kohta näemme, pitää vain kysyä kenttää sen Registry-avaimella ja sen jäkeen käyttää saatua arvoa tiedon tallentamiseen Rekisteriin.

Attribuutin määrittely

Huomaa edellä olevassa esimerkissä, että rakenne, jolla attribuutti liitetään tyyppiin tai jäseneen, näyttää hieman luokan instantioinnilta. Tämä johtuu siitä, että attribuutti on itse asiassa luokka, joka periytyy kantaluokasta *System.Attribute*.

Täydennetää nyt hieman *RegistryKey*-attribuuttia:

```
public enum RegistryHives
{
    HKEY_CLASSES_ROOT,
    HKEY_CURRENT_USER,
    HKEY_LOCAL_MACHINE,
    HKEY_USERS,
    HKEY_CURRENT_CONFIG
}

public class RegistryKeyAttribute : Attribute
{
    public RegistryKeyAttribute(RegistryHives Hive, String ValueName)
    {
        this.Hive = Hive;
        this.ValueName = ValueName;
    }

    protected RegistryHives hive;
    public RegistryHives Hive
    {
        get { return hive; }
        set { hive = value; }
    }

    protected String valueName;
    public String ValueName
    {
        get { return valueName; }
        set { valueName = value; }
    }
}
```

Olen lisännyt erilaisia Rekisterin avaimia varten luetellun tyypin (enum), muodostimen attribuuttiluokalle (joka saa parametrinaan Rekisterin avaimen ja nimen) ja kaksi ominaisuutta Rekisterin avainta ja nimeä varten. Voit tehdä paljon muutakin määritellessäsi attribuutteja, mutta tässä vaiheessa, kun tiedämme, miten attribuutteja

määritellään ja liitetään arvoja, jatketaan eteenpäin ja opiskellaan se, miten voimme kysellä attribuuttien arvoja suorituksen aikana. Siten saamme täysin toimivan esimerkin, jota voimme hyödyntää. Sen jälkeen siirrymme joihinkin edistyneempiin ominaisuuksiin liittyen attribuutin määrittelyyn ja liittämiseen.

> Huomaa Huomaa, että esimerkissä attribuuttiluokkien nimiin liitetään sana *Attribute*. Kun sitten liitän attribuutin tyyppiin tai jäseneen, en kuitenkaan käytä *Attribute*-jälkiliitettä. Tämä on oikotie, jonka meille tarjoavat C#-kääntäjän tekijät ilmaiseksi. Kun kääntäjä huomaa, että attribuutti liitetään tyyppiin tai jäseneen, se etsii *System.Attribute*-luokasta periytettyä luokkaa, jonka nimi on määritellyn attribuutin nimi. Jos luokkaa ei löydy, kääntäjä lisää sanan *Attribute* luokan nimeen ja etsii sitä. Siksi on yleinen tapa märitellä attribuuttiluokat päättymään sanaan *Attribute* ja sen jälkeen jättää se osa nimestä pois.

Attribuuttien kyseleminen

Tiedämme jo, miten määritellä attribuutti periyttämällä se *System.Attribute*-luokasta ja miten liittää se tyyppiin tai jäseneen. Mitä nyt? Miten voimme käyttää attribuutteja koodissa? Toisin sanoen, miten voimme kysyä tyypiltä tai jäseneltä siihen liitettyjä attribuutteja (ja sen parametreja)?

Kyselläksemme tyypiltä tai jäseneltä siihen liitettyjä attribuutteja, meidän pitää käyttää *reflection*-nimistä menetelmää. Se on edistynyt piirre, joka käydään läpi luvussa 16, "Metadatan kyseleminen reflection-metodien avulla," joten kerron tässä yhteydessä vain sen verran kuin on tarpeen tietojen kyselemiseen suorituksen aikana. Jos haluat tietää lisää asiasta, tutki lukua 16.

Reflection on menetelmä, jonka avulla voit suorituksen aikana dynaamisesti määritellä tyypin ominaisuudet. Voit esimerkiksi käyttää .NET Frameworkin *Reflection* APIa koko koosteen metadatan läpikäymiseen ja tuottaa luettelon kaikista luokista, tyypeistä ja metodeista, jotka tuossa koosteessa on määritelty. Katsotaan muutamia attribuuttiesimerkkejä ja miten niitä voidaan kysellä reflection-metodien avulla.

Luokan attribuutit

Miten haet attribuutin, joka on liitetty jäsentyyppiin? Sanotaan, että haluat määritellä attribuutin, joka kertoo sen etäpalvelimen nimen, jossa objekti luodaan. Ilman attribuutteja

tallentaisit tällaisen tiedon vakioon tai sovelluksen resurssitiedostoon. Attribuutin avulla voit liittää luokan etäpalvelimen nimitiedon itse luokkaan näin:

```
using System;

public enum RemoteServers
{
    JEANVALJEAN,
    JAVERT,
    COSETTE
}

public class RemoteObjectAttribute : Attribute
{
    public RemoteObjectAttribute(RemoteServers Server)
    {
        this.server = Server;
    }

    protected RemoteServers server;
    public string Server
    {
        get
        {
            return RemoteServers.GetName(typeof(RemoteServers),
                                         this.server);
        }
    }
}

[RemoteObject(RemoteServers.COSETTE)]
class MyRemotableClass
{
}
```

Määrittelet palvelimen, jossa objekti luodaan, käyttämällä seuraavan esimerkin mukaista koodia:

```
class ClassAttrApp
{
    public static void Main()
    {
        Type type = typeof(MyRemotableClass);
        foreach (Attribute attr in type.GetCustomAttributes())
        {
            RemoteObjectAttribute remoteAttr =
                attr as RemoteObjectAttribute;
```

(jatkuu)

```
            if (null != remoteAttr)
            {
                Console.WriteLine("Create this object on {0}.",
                                  remoteAttr.Server);
            }
        }
    }
}
```

Kuten voit olettaa, sovelluksen tuloste on tällainen:

```
Create this object on COSETTE.
```

Koska tämän esimerkki käyttää paljon yleistä koodia, tutkitaan mitä siinä liittyy reflection-menetelmään ja miten se palauttaa attribuutin arvon suorituksen aikana.

Ensimmäinen huomioitava rivi *Main*-metodissa on *typeof*-operaattorin käyttö:

```
Type type = typeof(MyRemotableClass);
```

Tämä operaattori palauttaa tyyppiin liitetyn *System.Type*-objektin, joka välitetään sille ainoana parametrina. Heti, kun sinulla on objekti, voit kysellä siltä tietoja.

Seuraavassa koodirivissä on kaksi selitettävää osaa:

```
foreach (Attribute attr in type.GetCustomAttributes(true))
```

Ensimmäinen on *Type.GetCustomAttributes*-metodin kutsu. Tämä metodi palauttaa *Attribute* -tyyppisen taulukon, joka tässä tapauksessa sisältää kaikki attribuutit, jotka on liitetty luokkaan nimeltä *MyRemotableClass*. Toinen selitettävä osa on *foreach*-käsky, jonka avulla käydään taulukko läpi täyttäen kunkin onnistuneen arvon *Attribute*-tyyppiseen muuttujaan *attr*.

Seuraava käsky käyttää *as*-operaattoria yrityksessä muuntaa *attr*-muuttuja tyypiksi *RemoteObjectAttribte*:

```
RemoteObjectAttribute remoteAttr =
        attr as RemoteObjectAttribute;
```

Sen jälkeen tutkimme null-arvoa, joka on *as*-operaattorin merkki epäonnistumisesta. Jos arvo ei ole null, eli että *remoteAttr*-muuttuja sisältää *MyRemotableClass*-tyyppiin liitetyn kelvollisen attribuutin, kutsumme yhtä *RemoteObjectAttribute*-ominaisuutta tulostaaksemme etäpalvelimen nimen:

```
if (null != remoteAttr)
{
    Console.WriteLine("Create this object on {0}",
                      remoteAttr.Server);
}
```

Metodin attribuutit

Nyt, kun olemme nähneet, miten luokan attribuutteja käsitellään, katsotaan metodin attribuuttien käyttämistä. Tämä tehdään erillisessä kappaleessa, koska reflection-koodi, joka metodin attribuutin kyselyyn tarvitaan, on erilainen kuin luokan attribuutin kyselyssä tarvittava. Tässä esimerkissä käytän attribuuttia, jota voidaan käyttää määrittämään, osallistuuko metodi tapahtumaan:

```
using System;
using System.Reflection;

public class TransactionableAttribute : Attribute
{
    public TransactionableAttribute()
    {
    }
}

class TestClass
{
    [Transactionable]
    public void Foo()
    {}

    public void Bar()
    {}

    [Transactionable]
    public void Baz()
    {}
}

class MethodAttrApp
{
    public static void Main()
    {
        Type type = Type.GetType(“TestClass”);
        foreach(MethodInfo method in type.GetMethods())
        {
            foreach (Attribute attr in
                     method.GetCustomAttributes())
            {
                if (attr is TransactionableAttribute)
                {
                    Console.WriteLine(“{0} is transactionable.",
                                      method.Name);
                }
```

(jatkuu)

```
            }
        }
    }
}
```

Koodin tuloste on seuraava:

```
Foo is transactionable.
Baz is transactionable.
```

Tässä nimenomaisessa esimerkissä pelkkä *TransactionableAttribute*-attribuutin olemassaolo on tarpeeksi ilmaisemaan, että metodi, jolle se kuuluu, voi ottaa osaa tapahtumaan. Siksi se on määritelty pelkkänä runkona, parametrittomana muodostimena ilman muita jäseniä.

TestClass-luokkaan määritellään kolme metodia, *Foo*, *Bar* ja *Baz*, joista kaksi (*Foo* ja *Baz*), on määritelty osallistumaan tapahtumaan. Huomaa, että kun liitetään attribuutti parametrittomaan muodostimeen, ei tarvitse lisätä avaavaa ja sulkevaa sulkumerkkiä.

Nyt hauskin osa. Katsotaan tarkemmin, miten voimme kysellä luokan metodilta sen attribuutteja. Aloitamme käyttämällä staattista *Type*-metodia *GetType* saadaksemme *TestClass*-luokan *System.Type*-objektin:

```
Type type = Type.GetType("TestClass");
```

Sen jälkeen käytämme *Type.GetMethods*-metodia hakiessamme taulukollisen *MethodInfo*-objekteja. Kukin näistä objekteista sisältää tietoja *TestClass*-luokan metodista. *foreach*-käskyllä käymme läpi jokaisen metodin:

```
foreach(MethodInfo method in type.GetMethods())
```

Nyt kun meillä on *MethodInfo*-objekti, voimme käyttää *MethodInfo.GetCustomAttributes*-metodia hakiessamme kaikki käyttäjän metodille luomat attribuutit. Kertauksen vuoksi, käytämme *foreach*-käskyä käydessämme läpi palautetun objektitaulukon:

```
foreach (Attribute attr in method.GetCustomAttributes(true))
```

Tässä vaiheessa koodia meillä on metodin attribuutti. Käyttämällä *is*-operaattoria, kysymme siltä, onko se *TransactionableAttribute*-attribuutti. Jos on, tulostamme metodin nimen:

```
if (attr is TransactionableAttribute)
{
Console.WriteLine("{0} is transactionable.",
                    method.Name);
}
```

Kentän attribuutit

Viimeisessä esimerkissämme katsomme, miten kysellään luokan kenttään liitettyjä attribuutteja. Sanotaan, että sinulla on luokka, joka sisältää muutamia kenttiä, joiden arvon haluat tallentaa Rekisteriin. Voit tehdä sen määrittelemällä attribuutin muodostimella, joka saa parametrikseen *enum*-tyypin arvon, joka esittää oikeaa Rekisterin avainta ja merkkijonon, joka esittää Rekisterin arvon nimeä. Voit sen jälkeen kysellä kentältä ohjelman suorituksen aikana sen Rekisterin avainta:

```
using System;
using System.Reflection;

public enum RegistryHives
{
    HKEY_CLASSES_ROOT,
    HKEY_CURRENT_USER,
    HKEY_LOCAL_MACHINE,
    HKEY_USERS,
    HKEY_CURRENT_CONFIG
}

public class RegistryKeyAttribute : Attribute
{
    public RegistryKeyAttribute(RegistryHives Hive, String ValueName)
    {
        this.Hive = Hive;
        this.ValueName = ValueName;
    }

    protected RegistryHives hive;
    public RegistryHives Hive
    {
        get { return hive; }
        set { hive = value; }
    }

    protected String valueName;
    public String ValueName
    {
        get { return valueName; }
        set { valueName = value; }
    }
}

class TestClass
```

(jatkuu)

```
{
    [RegistryKey(RegistryHives.HKEY_CURRENT_USER, “Foo”)]
    public int Foo;

    public int Bar;
}

class FieldAttrApp
{
    public static void Main()
    {
        Type type = Type.GetType(“TestClass”);
        foreach(FieldInfo field in type.GetFields())
        {
            foreach (Attribute attr in field.GetCustomAttributes())
            {
                RegistryKeyAttribute registryKeyAttr =
attr as RegistryKeyAttribute;
                if (null != registryKeyAttr)
                {
                    Console.WriteLine
                        (“{0} will be saved in {1}\\\\{2}",
                         field.Name,
                         registryKeyAttr.Hive,
                         registryKeyAttr.ValueName);
                }
            }
        }
    }
}
```

En käy kaikkea koodia läpi, koska osa siitä on samaa kuin edellisessä esimerkissä. Muutama tärkeä seikka kuitenkin kannattaa painaa mieleen. Huomaa ensinnäkin, että samoin kuin *MethodInfo*-objekti on määritelty hakemaan tietoja tyyppiobjektilta, tarjoaa *FieldInfo*-objekti saman toiminnallisuuden kentän tietojen hakemiseen tyyppiobjektilta. Kuten edellisessä esimerkissäkin, aloitamme selvittämällä testiluokkaamme liitetyn tyyppiobjektin. Sen jälkeen käymme läpi *FieldInfo*-taulukon ja kullakin *FieldInfo*-objektilla käymme läpi sen atribuutit, kunnes löydämme etsimämme eli *RegistryKeyAttribute*-attribuutin. Jos ja kun löydämme sen, tulostamme kentän nimen ja haemme attribuutilta sen *Hive* ja *ValueName* -kentät.

Attribuutin parametrit

Edellä olleessa esimerkissä kuvasin liitettyjen attribuuttien käyttö niiden muodostimien kautta. Nyt kerron muutamasta attribuutin muodostimiin liittyvästä asiasta, joista ei ole aiemmin ollut puhetta.

Sijaintiparametrit ja nimetyt parametrit

FieldAttrApp-esimerkissä edellisessä kappaleessa näit *RegistryKeyAttribute*-nimisen attribuutin. Sen muodostin näytti seuraavalta:

public RegistryKeyAttribute(RegistryHives *Hive*, String *ValueName*)

Muodostimen rakenteen perusteella attribuutti liitetään kenttään seuraavasti:

```
[RegistryKey(RegistryHives.HKEY_CURRENT_USER, "Foo")]
public int Foo;
```

Tämä on selvää ja suoraviivaista. Muodostimella on kaksi parametria ja kahta parametria käytetään sen liittämiseen kenttään. Voimme tehdä tämän kuitenkin helpommaksi ohjelmoijan kannalta. Jos parametri pysyy muuttumattomana suurimman osan aikaa, miksi selvittää luokan käyttäjän tyyppi joka kerta? Voimme asettaa oletusarvon käyttämällä sijantiparametreja ja nimettyjä parametreja.

Sijaintiparametrit ovat paramatreja attribuutin muodostimelle. Ne ovat pakollisia ja ne pitää määritellä aina kun attribuuttia käytetään. Yllä olevassa *RegistryKeyAttribute*-esimerkissä ovat sekä *Hive* että *ValueName* sijaintiparamatreja. Nimettyjä parametreja ei varsinaisesti määritellä attribuutin muodostimessa vaan ne ovat ei-staattisia kenttiä ja ominaisuuksia. Siksi nimetyt parametrit antavat asiakasohjelmalle mahdollisuuden asettaa attribuutin kentät ja ominaisuudet, kun attribuutti instantioidaan ilman, että jokaista mahdollista kenttien ja ominaisuuksien yhdistelmää varten, joita asiakasohjelma mahdollisesti haluaa käyttää, pitää kirjoittaa muodostimet.

Kukin julkinen muodostin voi määrittää sijaintiparametrien järjestyksen. Tämä on sama kuin minkä tahansa tyyppisellä luokalla. Attribuuttien tapauksessa kuitenkin, kun sijaintiparametrit on sijoitettu, käyttäjä voi viitata määrättyihin kenttiin tai ominaisuuksiin merkintätavalla *FieldOrPropertyName=Value*. Muokataan *RegistryKeyAttribute*-attribuutti tällaiseksi. Tässä esimerkissä teemme *RegistryKeyAttribute.ValueName*-parametrista ainoan sijaintiparametrin ja *RegistryKeyAttribute.Hive*-attribuutista ainoan valinnaisen parametrin. Kysymys kuuluu nyt, ”Miten määrittelet jotakin valinnaiseksi nimetyksi parametriksi?” Koska vain sijaintiparametrit (ja siis pakolliset) sisältyvät muodostimen määrittelyyn, ota ne yksinkertaisesta pois sieltä. Käyttäjä voi sitten viitata nimettynä parametrina jokaiseen

kenttään joka ei ole *readonly*, *static* tai *const* ja jokaiseen ei-staattiseen ominaisuuteen, jolla on käsittelymetodi arvon asettamiseen. Saadaksemme siis *RegistryKeyAttribute.Hive*-parametrista nimetyn parametrin, poistamme sen muodostimen määrittelystä, koska se on jo yleinen (public) read/write-ominaisuus:

```
public RegistryKeyAttribute(String ValueName)
```

Käyttäjä voi nyt liittää attribuutin jommallakummalla seuraavista tavoista:

```
[RegistryKey(“Foo”)]
[RegistryKey("Foo", Hive = RegistryHives.HKEY_LOCAL_MACHINE)]
```

Saat joustavuutta antamalla kentälle oletusarvon ja samalla annat käyttäjälle mahdollisuuden korvata sen tarpeen mukaan. Mutta hetkinen! Jos käyttäjä ei aseta *RegistryKeyAttribute.Hive*-kentän arvoa, miten annamme sille oletuasarvon? Saatat ajatella ratkaisevasi sen näin: ”Tarkistan, onko se asetettu muodostimessa.” Ongelma on kuitenkin se, että *RegistryKeyAttribute.Hive* on lueteltu tyyppi, jonka taustatyyppi on int eli se on arvotyyppi. Tämä tarkoittaa, että kääntäjä oletuksena alustaa sen arvoon 0! Jos tutkimme *RegistryKeyAttribute.Hive*-parametrin arvoa muodostimessa ja huomaamme, että se on nolla, emme tiedä onko sen asettanut kutsuva ohjelma nimetyn parametrin avulla vai onko kääntäjä alustanut sen, koska se on arvotyyppi. Valitettavasti tällä hetkellä ainoa tapa, jonka tiedän tämän ongelman kiertämiseksi, on muuttaa koodia niin, että arvo 0 ei ole kelvollinen. Se voidaan tehdä muuttamalla RegistryHives *enum*-lueteltujen tyyppien arvoja seuraavasti:

```
public enum RegistryHives
{
    HKEY_CLASSES_ROOT = 1,
    HKEY_CURRENT_USER,
    HKEY_LOCAL_MACHINE,
    HKEY_USERS,
    HKEY_CURRENT_CONFIG
}
```

Nyt tiedämme, että ainoa tapa, jolla *RegistryKeyAttribute.Hive*-parametri voi olla nolla, on silloin, kun kääntäjä on alustanut sen nollaksi eikä käyttäjä ole korvannut sitä nimetyn parametrin avulla. Voimme nyt kirjoittaa seuraavan koodin sen alustamiseen:

```
    public RegistryKeyAttribute(String ValueName)
    {
        if (this.Hive == 0)
            this.Hive = RegistryHives.HKEY_CURRENT_USER;

        this.ValueName = ValueName;
    }
```

Nimettyihin parametreihin liittyviä virheitä

Kun käytetään nimettyjä paramatreja, sinun tulee määritellä sijaintiparametrit ensin. Sen jälkeen nimetyt parametrit voivat olla missä järjestyksessä tahansa, koska ne tunnistetaan kentän tai ominaisuuden nimellä. Esimerkiksi seuraava koodi antaa kääntäjän virheen:

```
// Tämä on virhe, koska sijaintiparametrit eivät
// voi olla nimettyjen parametrien jälkeen.
[RegistryKey(Hive=RegistryHives.HKEY_LOCAL_MACHINE, “Foo”)]
```

Lisäksi et voi nimetä sijaintiparametreja. Kun kääntäjä ratkaisee attribuutin käyttöä, se yrittää ratkaista nimetyt parametrit ensin. Sen jälkeen se yrittää ratkaista kaiken, mitä jäi metodin parametriluettelosta jäljelle, eli sijaintiparametrit. Seuraava koodi ei käänny, koska kääntäjä voi ratkaista jokaisen nimetyn parametrin, mutta kun ne ovat loppuneet, se ei löydä yhtään sijaintiparametria ja ilmoittaa “*No overload for method 'RegistryKeyAttribute' takes '0'‘ arguments*”:

```
[RegistryKey(ValueName="Foo", Hive=RegistryHives.HKEY_LOCAL_MACHINE)]
```

Lopuksi, nimetty parametri voi olla jokainen julkisesti käsiteltävä kenttä tai ominaisuus, mukaan lukien setter-metodi, joka ei ole staattinen tai vakio eli sen käsittelymääre ei ole *static* tai *const*.

Attribuutin kelvolliset parametrityypit

Attribuuttiluokan sijaintiparametrit ja nimetyt parametrit voivat olla vain attribuuttiparametrityyppisiä, joita ovat seuraavat:

- *bool*, *byte*, *char*, *double*, *float*, *int*, *long*, *short*, *string*
- *System.Type*
- *object*
- Lueteltu tyyppi (*enum*) niin, että se ja jokainen siihen sisältyvä tyyppi, on julkisesti käsiteltävissä, kuten edellä olleessa esimerkissä oli asia
- Yksiulotteinen taulukko, jonka sisältö on jokin edellä luetelluista tyypeistä

Koska parametrien tyypit on rajattu edellä olevaan luetteloon, et voi välittää luokan tapaisia tietorakenteita attribuutin muodostimelle. Tässä rajoituksessa on järkeä, koska attribuutit liitetään suunnittelun aikana eikä sinulla ole luokan instanssia (eli objektia) silloin. Yllä lueteltujen kelvollisten tyyppien avulla voit kovakoodata niiden arvot suunnittelun aikana ja sen ansiosta niitä voidaan käyttää.

AttributeUsage-attribuutti

Tavallisiin C#-tyyppeihin liittämiesi omien attribuuttiesi lisäksi voit käyttää *AttributeUsage*-attribuuttia määrittämään, miten haluat näitä attribuutteja käytettävän. *AttributeUsage*-attribuutilla on seuraava dokumentoitu kutsumuoto:

```
[AttributeUsage(
validon,
AllowMultiple = allowmultiple,
Inherited = inherited
)]
```

Kuten näet, on helppo huomata, mitä ovat sijaintiparametreja ja mitkä nimettyjä parametreja. Suosittelen, että dokumentoit attribuuttisi tällä tavalla, jotta attribuuttisi käyttäjien ei tarvitse tutkia attribuuttiluokkasi lähdekoodia etsiessään julkisia read/write-kenttiä ja ominaisuuksia, joita voidaan käyttää nimettyinä parametreina.

Attribuutin kohteen määrääminen

Katsotaan nyt uudelleen edellisen kappaleen *AttributeUsage*-attribuuttia. Sen *validon*-parametri on sijaintiparametri, eli siksi pakollinen. Tämän parametrin avulla voit määritellä ne tyypit, joihin attribuuttisi voidaan liittää. Itse asiassa *AttributeUsage*-attribuutin *validon*-parametrin tyyppi on *AttributeTargets*, joka on seuraavalla tavalla määritelty lueteltu tyyppi (enun):

```
public enum AttributeTargets
{
   Assembly    = 0x0001,
   Module      = 0x0002,
   Class       = 0x0004,
   Struct      = 0x0008,
   Enum        = 0x0010,
   Constructor = 0x0020,
   Method      = 0x0040,
   Property    = 0x0080,
   Field       = 0x0100,
   Event       = 0x0200,
   Interface   = 0x0400,
   Parameter   = 0x0800,
   Delegate    = 0x1000,
   All = Assembly | Module | Class | Struct | Enum | Constructor |
         Method | Property | Field | Event | Interface | Parameter |
         Delegate,
```

```
   ClassMembers  =  Class | Struct | Enum | Constructor | Method |
                    Property | Field | Event | Delegate | Interface,
}
```

Huomaa, että käyttämällä *AttributeUsage*-attribuuttia, voit määritellä *Attributetargets.All*, jolloin attribuutti voidaan liittää mihin tahansa *AttributeTargets*-luettelossa määriteltyyn tyyppiin. Tämä on myös oletus, jos et määrittele *AttributeUsage*-attribuuttia ollenkaan. Koska tämä *AttributeTargets.All* on oletusarvo, saatat miettiä, miksi *validon*-parametria tulisi käyttää ylipäänsä. Tällä attribuutilla voit käyttää nimettyjä parametreja ja haluat ehkä muuttaa jotain niitä. Muista, että jos käytät nimettyjä parametreja, sinun pitää kirjoittaa sen eteen kaikki sijaintiparametrit. Täten sinulla on helppo tapa ilmoittaa, että haluat oletusattribuutille käyttömahdollisuuden *AttributeTargets.All* ja voit silti asettaa nimettyjä parametreja.

Milloin tulisi määritellä *validon*-parametri (*AttributeTarget*-tyyppisen) ja miksi? Käytät sitä, kun haluat määrätä attribuutin käytön täsmällisesti. Yllä olevissa esimerkeissä loimme *RemoteObjectAttribute*-attribuutin, jota voidaan soveltaa vain luokille, *TransactionableAttribute*-attribuution, jota voidaan soveltaa vain metodeihin ja *RegistryKeyAttribute*-attribuutin, joka on järkevä vain kentillä. Jos haluaisimme varmistaa, että näitä attribuutteja käytetään vain liitettynä tyyppeihin, joita varten ne on suunniteltu, voisimme määritellä ne seuraavasti (attribuuttien runko on jätetty selvyyden vuoksi pois):

```
[AttributeUsage(AttributeTargets.Class)]
public class RemoteObjectAttribute : Attribute]
{
    ...
}

[AttributeUsage(AttributeTargets.Method)]
public class TransactionableAttribute : Attribute
{
    ...
}

[AttributeUsage(AttributeTargets.Field)]
public class RegistryKeyAttribute : Attribute
{
    ...
}
```

Viimeinen seikka liittyy lueteltuun tyyppiin *AttributeTargets*: Voit yhdistää niitä käyttämällä |-operaattoria. Jos sinulla on attribuutti, jota voidaan soveltaa sekä kenttiin että ominaisuuksiin, voit määritellä sen *AttributeUsage*-attribuutin seuraavasti:

```
[AttributeUsage(AttributeTargets.Field | AttributeTargets.Property)]
```

Yksikäyttöiset ja monikäyttöiset attribuutit

Voit määritellä attribuutin yksi- tai monikäyttöiseksi *AttributeUsage*-attribuutin avulla. Tämä määrittely määrää, miten monta kertaa yhtä attribuuttia voidaan käyttää yhdessä kentässä. Oletuksena kaikki attribuutit ovat yksikäyttöisiä, jolloin seuraavan koodin kääntäminen aiheuttaa virheilmoituksen:

```
using System;
using System.Reflection;

public class SingleUseAttribute : Attribute
{
    public SingleUseAttribute(String str)
    {
    }
}

// VIRHE: Tämä  aiheuttaa “duplicate attribute” -virheen.
[SingleUse(“abc”)]
[SingleUse(“def”)]
class MyClass
{
}

class SingleUseApp
{
    public static void Main()
    {
    }
}
```

Tämä ongelma ratkeaa niin, että määrittelet *AttributeUsege*-attribuutin monikäyttöiseksi, jolloin se voidaan liittää tyyppiin useita kertoja. Seuraava koodi toimii:

```
using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.All, AllowMultiple=true)]
public class SingleUseAttribute : Attribute
{
    public SingleUseAttribute(String str)
    {
    }
}

[SingleUse(“abc”)]
[SingleUse(“def”)]
class MyClass
{
```

```
}

class MultiUseApp
{
    public static void Main()
    {
    }
}
```

Käytännön esimerkki tilanteesta, jossa voit käyttää tätä tapaa, on *RegistryKeyAttribute*-attribuutin kanssa, josta puhuttiin kappaleessa ”Attribuutin määrittely.” Koska on mahdollista, että kenttä voidaan tallentaa useaan paikkaan Rekisterissä, sinun ehkä kannattaa liittää *AttributeUsage*-attribuuttiin nimetty parametri *AllowMultiple*, kuten edellä olleessa koodissa.

Attribuutin periytymisen määrittely

Viimeinen *AttributeUsage*-attribuutin parametri on *inherited*-lippu, joka ilmoittaa, voidaanko attribuutti periyttää. Sen oletusarvo on *false*. Jos *inherited*-lippu on asetettu *true*ksi, sen merkitys riippuu *AllowMultiple*-parametrin arvosta. Jos *inherited* on *true* ja *AllowMultiple* on *false*, attribuutti voi korvata perityn attribuutin. Mutta jos *inherited*-lippu on *true* ja *AllowMultiple* on myös *true*, attribuutti kerääntyy jäseneen.

Attribuutin tunniste

Katso seuraavaa koodiriviä ja yritä selvittää, liittyykö attribuutti paluuarvoon vai metodiin:

```
class MyClass
{
    [HRESULT]
    public long Foo();
}
```

Jos sinulla on COM-kokemusta, tiedät, että HRESULT on kaikkien metodien paluuarvon tyyppi, lukuunottamatta *AddRef* ja *Release*-nimisiä metodeja. On kuitenkin helppo huomata, että jos attribuutin nimi voidaan liittää sekä paluuarvoon että metodin nimeen, on kääntäjän mahdotonta tietää, kumpaa haluat. Seuraavassa muutamia tilanteita, joissa kääntäjä ei voit tietää tarkoitustasi koodin perusteella:

- metodi - paluuarvo
- tapahtuma - kenttä - ominaisuus

- delegaatti - paluuarvo
- ominaisuus - käsittelijä - getter-metodin paluuarvo - setter-metodin parametri

Kussakin näissä tapauksissa kääntäjä tekee päätöksensä ”yleisyyden” mukaan. Voit korvata tämän päätöksen käyttämällä attribuutin tunnistetta, jotka ovat kaikki seuraavassa luettelossa.

- assembly
- module
- type
- method
- property
- event
- field
- param
- return

Attribuutin tunnistetta käytetään kirjoittamalla se puolipisteellä erotettuna attribuutin nimen eteen. Jos haluat *MyClass*-esimerkissä varmistaa, että kääntäjä ymmärtää, että HRESULT-attribuutti on tarkoitettu paluuarvolle eikä metodille, voit määritellä sen seuraavasti:

```
class MyClass
{
    [return:HRESULT]
    public long Foo();
}
```

Yhteenveto

C#:n attribuutit tarjoavat menetelmän, jolla liitetään tyyppejä ja jäseniä suunnittelun aikana informaatioon, joka voidaan myöhemmin suorituksen aikana hakea *reflection*-menetelmällä. Attribuuttien avulla voit tehdä täysin itsenäisiä, itsensä kuvaavia komponentteja eikä sinun tarvitse kirjoittaa tarpeellisia tietoja resurssitiedostoihin ja vakioihin. Etuna saadaan siirrettävämpiä komponentteja, joita on helpompi kirjoittaa ja ylläpitää.