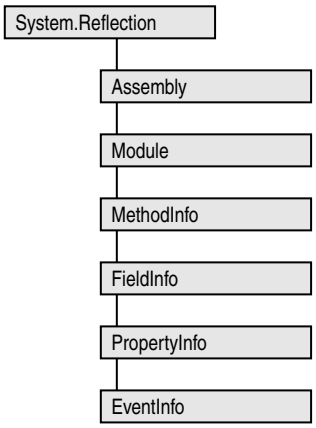
16

Metadatan kyseleminen Reflection-metodeilla

Luvussa 2, ”Johdanto Microsoft .NETiin,” kerroin, miten kääntäjä generoi Win32 Portable Executable (PE)-tiedoston, joka sisältää MSIL:n ja metadatan. Yksi .NETin tehokas ominaisuus on se, että voit kirjoittaa koodia, joka käsittelee sovelluksen metadataa *reflection*-nimisen toiminnon avulla. Yksinkertaisesti sanoen, reflection on kyky selvittää tyypin informaatio suorituksen aikana. Tässä luvussa käsittelen reflection-API:n ja miten käyt sen avulla läpi koosteen moduulit ja tyypit ja haet erilaisia tyypin suunnitteluaikaisia ominaisuuksia. Näet myös muutamia kehittyneempiä reflection-menetelmän käyttätapoja, kuten metodien dynaamisen käynnistämisen ja tyyppi-informaation käyttämisen myöhäisessä sidonnassa ja jopa MSIL-koodin luomisen ja suorittamisen suorituksen aikana!

Reflection-API:n rakenne

.NET Reflection API on joukko *System.Reflection*-nimiavaruudessa määriteltyjä luokkia. Osan luokista näet kuvassa 16-1. Näiden luokkien avulla voit selvittää koosteen ja tyypin tietoja. Voit aloittaa mistä tahansa hierarkkian kohtaa sovelluksesi tarpeet vaativat.



Kuva 16-1 Osa .NETin *System.Reflection*-luokkahierarkkiaa.

Nämä luokat käsittävät suurimman osan *System.Reflection* -luokkien toiminnallisuudesta. En aio luetella jokaisen luokan jokaista metodia ja kenttää, vaan esitän tärkeimpien luokkien perusteet ja näytän esimerkkiohjelman, joka sisältää ne toiminnot, joita luultavimmin tulet omissa sovelluksissasi tarvitsemaan.

Type-luokka

Koko toiminnon ytimenä on *System.Type*-luokka. Se on abstrakti luokka, joka esittää Common Type System (CTS):n tyyppiä. Sen avulla voit kysyä tyypin nimen, tyypin sisältämät modulit ja nimiavaruudet ja sen, onko tyyppi arvotyyppi vain viittaustyyppi.

Instanssin tyypin selvittäminen

Seuraava esimerkki näyttää, miten haet instantioidun *int*-tyypin *Type*-objektin:

```
using System;
using System.Reflection;

class TypeObjectFromInstanceApp
{
    public static void Main(string[] args)
    {
        int i = 6;
        Type t = i.GetType();
        Console.WriteLine(t.Name);
    }
}
```

Type-objektin hakeminen nimen perusteella

Sen lisäksi, että voit hakea muuttujan *Type*-objektin, voit myös luoda *Type*-objektin tyypin nimen avulla. Toisin sanoen, sinulla ei tarvitse olla tyypin instanssia. Tässä esimerkki, joka tekee tämän *System.Int32*-tyypille:

```
using System;
using System.Reflection;

class TypeObjectFromNameApp
{
    public static void Main(string[] args)
    {
        Type t = Type.GetType("System.Int32");
        Console.WriteLine(t.Name);
    }
}
```

Huomaa, että et voi käyttää C#:n peitenimiä, kun kutsut *Type.GetType*-metodia, koska sitä käytetään kaikissa kielissä Siksi et voi käyttää C#:n *int*-peitenimeä *System.Int32*:n sijalla.

Tyyppien tulkitseminen

Voit *System.Type*-luokan avulla myös kysyä tyypiltä sen lähes kaikkia attribuutteja, näiden käsittelymääreitä ja ovatko ne sisäkkäisiä, sen COM-ominaisuuksia ja niin edelleen. Seuraava koodi näyttää tämän. Käytän siinä useita tavallisia tyyppejä sekä muutamia omia tyyppejä:

```
using System;
using System.Reflection;

interface DemoInterface
{
}

class DemoAttr : System.Attribute
{
}

enum DemoEnum
{
}

public class DemoBaseClass
{
}
```

(jatkuu)

```
public class DemoDerivedClass : DemoBaseClass
{
}

class DemoStruct
{
}

class QueryTypesApp
{
    public static void QueryType(string typeName)
    {
        try
        {
            Type type = Type.GetType(typeName);
            Console.WriteLine("Type name: {0}", type.FullName);
            Console.WriteLine("\tHasElementType = {0}",
                              type.HasElementType);
            Console.WriteLine("\tIsAbstract = {0}", type.IsAbstract);
            Console.WriteLine("\tIsAnsiClass = {0}", type.IsAnsiClass);
            Console.WriteLine("\tIsArray = {0}", type.IsArray);
            Console.WriteLine("\tIsAutoClass = {0}", type.IsAutoClass);
            Console.WriteLine("\tIsAutoLayout = {0}",
                              type.IsAutoLayout);
            Console.WriteLine("\tIsByRef = {0}", type.IsByRef);
            Console.WriteLine("\tIsClass = {0}", type.IsClass);
            Console.WriteLine("\tIsCOMObject = {0}", type.IsCOMObject);
            Console.WriteLine("\tIsContextful = {0}",
                              type.IsContextful);
            Console.WriteLine("\tIsEnum = {0}", type.IsEnum);
            Console.WriteLine("\tIsExplicitLayout = {0}",
                              type.IsExplicitLayout);
            Console.WriteLine("\tIsImport = {0}", type.IsImport);
            Console.WriteLine("\tIsInterface = {0}",
                              type.IsInterface);
            Console.WriteLine("\tIsLayoutSequential = {0}",
                              type.IsLayoutSequential);
            Console.WriteLine("\tIsMarshalByRef = {0}",
                              type.IsMarshalByRef);
            Console.WriteLine("\tIsNestedAssembly = {0}",
                              type.IsNestedAssembly);
            Console.WriteLine("\tIsNestedFamANDAssem = {0}",
                              type.IsNestedFamANDAssem);
            Console.WriteLine("\tIsNestedFamily = {0}",
                              type.IsNestedFamily);
            Console.WriteLine("\tIsNestedFamORAssem = {0}",
```

```
                        type.IsNestedFamORAssem);
            Console.WriteLine("\tIsNestedPrivate = {0}",
                        type.IsNestedPrivate);
            Console.WriteLine("\tIsNestedPublic = {0}",
                        type.IsNestedPublic);
            Console.WriteLine("\tIsNotPublic = {0}",
                        type.IsNotPublic);
            Console.WriteLine("\tIsPointer = {0}",
                        type.IsPointer);
            Console.WriteLine("\tIsPrimitive = {0}",
                        type.IsPrimitive);
            Console.WriteLine("\tIsPublic = {0}",
                        type.IsPublic);
            Console.WriteLine("\tIsSealed = {0}",
                        type.IsSealed);
            Console.WriteLine("\tIsSerializable = {0}",
                        type.IsSerializable);
            Console.WriteLine("\tIsServicedComponent = {0}",
                        type.IsServicedComponent);
            Console.WriteLine("\tIsSpecialName = {0}",
                        type.IsSpecialName);
            Console.WriteLine("\tIsUnicodeClass = {0}",
                        type.IsUnicodeClass);
            Console.WriteLine("\tIsValueType = {0}", type.IsValueType);
        }
        catch(System.NullReferenceException)
        {
            Console.WriteLine
                        ("{0} is not a valid type", typeName);
        }
    }

        public static void Main(string[] args)
        {
                QueryType("System.Int32");
                QueryType("System.Int64");
                QueryType("System.Type");

                QueryType("DemoAttr");
                QueryType("DemoEnum");
                QueryType("DemoBaseClass");
                QueryType("DemoDerivedClass");
                QueryType("DemoStruct");
        }
    }
```

Työskentely koosteilla ja moduleilla

Koosteista puhutaan tarkemmin luvussa 18, ”Työskentely koosteilla.” Tämän luvun asiaa varten kerron, että kooste on fyysinen tiedosto, joka sisältää useita .NET PE -tiedostoja. Koosteen suurin etu on siinä, että voit sen avulla koota yhteen joukon toimintoja helpottamaan jakelua ja versiointia. .NETin koosteen ajonaikainen ympäristö (ja reflection-luokkahierarkkian kantaluokka) on *Assembly*-luokka.

Assembly-luokalla voit tehdä monia asioita. Seuraavana on muutamia yleisempiä tehtäviä, joita katsomme kohta tarkemmin:

- Koosteen tyyppien selvittäminen
- Luettelo koosteen moduleista
- Tunnistusinformaation (koosteen fyysinen nimi ja sijainti) määrittäminen
- Versiointi ja turva-informaation tulkinta
- Koosteen aloituspisteen hakeminen

Koosteen tyyppien selvittäminen

Määrätyn koosteen sisältämien tyyppien selvittäminen onnistuu siten, että instantioit *Assembly*-objektin ja pyydät haluamasi koosteen *Types*-taulukkoa. Tässä esimerkki:

```
using System;
using System.Diagnostics;
using System.Reflection;

class DemoAttr : System.Attribute
{
}

enum DemoEnum
{
}

class DemoBaseClass
{
}
```

```
class DemoDerivedClass : DemoBaseClass
{
}

class DemoStruct
{
}

class GetTypesApp
{
    protected static string GetAssemblyName(string[] args)
    {
        string assemblyName;

        if (0 == args.Length)
        {
            Process p = Process.GetCurrentProcess();
            assemblyName = p.ProcessName + “.exe";
        }
        else
            assemblyName = args[0];

        return assemblyName;
    }

    public static void Main(string[] args)
    {
        string assemblyName = GetAssemblyName(args);

        Console.WriteLine(“Loading info for “ + assemblyName);
        Assembly a = Assembly.LoadFrom(assemblyName);

        Type[] types = a.GetTypes();
        foreach(Type t in types)
        {
            Console.WriteLine(“\nType information for: “ +
                              t.FullName);
            Console.WriteLine(“\tBase class = “ +
                              t.BaseType.FullName);
        }
    }
}
```

> Huomaa Jos yrität suorittaa intranetin kautta koodia, joka tarvitsee turvaselvityksen (kuten reflection-APIa käyttävä koodi), sinun pitää muokata järjestelmän asetustiedostoa. Yksi tapa tehdä se on käyttää Code Access Security Policy (caspol.exe) -apuohjelmaa. Tässä esimerkki sen käytöstä:
>
> ```
> caspol -addgroup 1.2 -url “file://somecomputer/someshare/
> *" SkipVerification
> ```
>
> Tämä esimerkki myöntää lisäoikeuden (tässä tapauksessa SkipVerification-oikeuden) sen URL:n perusteella, josta koodi suoritetaan. Voit myös muokata kerralla kaikkia saman vyöhykkeen koodien oikeuksia tai vaikkapa vain yhden koosteen oikeuksia. Näet caspol.exe:n kaikki käyttövaihtoehdot kirjoittamalla käskyriville caspol -? tai tutkimalla MSDN-online-dokumentteja.

Main-metodin alku ei ole erityisen mielenkiintoinen. Siinä määritellään, oletko syöttänyt sovellukselle koosteen nimen. Jos et, käytetään *Process*-luokan staattista *GetProcessName*-metodia hakemaan käynnissä olevan sovelluksen nimi.

Sen jälkeen huomaat, miten helppoa useimmat reflection-tehtävät ovat. Helpoin tapa instantioida *Assembly*-objekti on kutsua *Assembly.LoadFrom*-metodia. Se ottaa yhden parametrin: merkkijonon, joka kertoo sen fyysisen tiedoton nimen, jonka haluat ladata. Sen jälkeen *Assembly.GetTypes*-metodin kutsu palauttaa taulukon *Type*-objekteja. Tässä vaiheessa meillä on objekti, joka kuvaa koko koosteen jokaisen yksittäisen tyypin. Lopuksi sovellus tulostaa kantaluokkansa.

Tässä tuloste sovelluksen suorittamisesta, kun joko määrittelit *gettypes.exe*-tiedoston nimen tai et välittänyt sovellukselle lainkaan parametria:

```
Loading info for GetTypes.exe

Type information for: DemoAttr
    Base class = System.Attribute

Type information for: DemoEnum
    Base class = System.Enum

Type information for: DemoBaseClass
    Base class = System.Object
```

```
Type information for: DemoDerivedClass
    Base class = DemoBaseClass

Type information for: DemoStruct
    Base class = System.Object

Type information for: AssemblyGetTypesApp
    Base class = System.Object
```

Koosteen modulien luettelo

Vaikka useimmat tämän kirjan sovelluksista koostuvat yhdestä modulista, voit luoda myös koosteita, jotka sisältävät useita moduleja. Voit hakea koosteen modulit kahdella tavalla. Ensimmäinen on pyytää modulit sisältävä taulukko. Sen avulla voit käydä ne kaikki läpi ja hakea tarvitsemasi tiedot. Toinen tapa on hakea määrätty moduli. Katsotaan kumpaakin vaihtoehtoa.

Jotta voisin käydä koosteen modulit läpi, minulla pitää olla kooste, jossa on enemmän kuin yksi moduuli. Teen sellaisen siirtämällä *GetAssemblyName*-metodin omaan luokkaansa ja sijoittamalla sen *AssemblyUtil.netmodule*-nimiseen tiedostoon, tällä tavalla:

```
using System.Diagnostics;

namespace MyUtilities
{
    public class AssemblyUtils
    {
        public static string GetAssemblyName(string[] args)
        {
            string assemblyName;

            if (0 == args.Length)
            {
                Process p = Process.GetCurrentProcess();
                assemblyName = p.ProcessName + ".exe";
            }
            else
                assemblyName = args[0];

            return assemblyName;
        }
    }
}
```

Moduli luodaan sitten seuraavasti:

```
csc /target:module AssemblyUtils.cs
```

/target:module-asetus määrää kääntäjän tekemään modulin, joka myöhemmin sijoitetaan koosteeseen. Yllä oleva käskyrivi luo *AssemblyUtil.netmodule*-nimisen tiedoston. Luvussa 18 kerron yksityiskohtaisesti eri asetuksista koosteiden ja modulien luonnissa.

Tässä vaiheessa tarvitsen vielä toisen modulin, jotta meillä on jotain tutkittavaa. Seuraavassa sovellus, joka käyttää *AssemblyUtils*-luokkaa. Huomaa *using*-määreen käyttö, kun viitataan *MyUtilities*-nimiavaruuteen.

```
using System;
using System.Reflection;
using MyUtilities;

class GetModulesApp
{
    public static void Main(string[] args)
    {
        string assemblyName = AssemblyUtils.GetAssemblyName(args);

        Console.WriteLine("Loading info for " + assemblyName);
        Assembly a = Assembly.LoadFrom(assemblyName);

        Module[] modules = a.GetModules();
        foreach(Module m in modules)
        {
            Console.WriteLine("Module: " + m.Name);
        }
    }
}
```

Tämän sovelluksen kääntäminen ja *AssemblyUtils.netmodule*-modulin lisääminen samaan koosteeseen tapahtuu seuraavanlaisella käskyrivillä:

```
csc /addmodule:AssemblyUtils.netmodule GetModules.cs
```

Nyt sinulla on kooste, jossa on kaksi erilaista modulia. Näet sen, kun käynnistät sovelluksen. Tulos näyttää tältä:

```
Loading info for GetModulesApp.exe
Module: GetModulesApp.exe
Module: AssemblyUtils.netmodule
```

Kuten näet koodista, yksinkertaisesti instantioin *Assembly*-objektin ja kutsuin sen *GetModules*-metodia. Sitten käyn paluutaulukon läpi ja tulostan kunkin modulin nimen.

Myöhäinen sidonta Reflection-menetelmän avulla

Muutama vuosi sitten työskentelin IBM:n multimediayksikössä, sen IBM/World Book Multimedia Encyclopedia -tuotteen parissa. Haaste, jonka kohtasimme sovelluksessa, aiheutui siitä, että halusimme antaa käyttäjälle mahdollisuuden asentaa erilaisia tiedonsiirtoprotokollia World Book -palvelimille. Ratkaisun tuli olla dynaaminen, jotta käyttäjä voisi tilanteen mukaan lisätä ja poistaa eri protokollia (esimerkiksi TCP/IP, IGN, CompuServe ja niin edelleen) järjestelmästään. Sovelluksen piti toisaalta tietää, mitä protokollia oli käytettävissä, jotta hän pystyi valitsemaan niistä jonkun käyttöön. Päädyimme ratkaisuun, jossa loimme määrätyn tarkenteen omaavia DLL:iä ja asensimme ne sovelluksen kansioon. Kun käyttäjä halusi nähdä luettelon asennetuista protokollista, sovellus kutsui Win32:n *LoadLibrary*-funktiota ladatakseen kunkin DLL:n ja kutsui sitten *GetProcAddress*-funktiota hankkiakseen funktio-osoittimen haluttuun funktioon. Tämä on täydellinen esimerkki myöhäisestä sidonnasta tavallisessa Win32-ohjelmoinnissa, jossa kääntäjä ei käännöksen aikana tiedä näistä kutsuista mitään. Kuten näet seuraavassa esimerkissä, sama tehtävä voidaan .NETissä toteuttaa käyttämällä *Assembly*-luokkaa, tyypin selvittämistä ja uutta *Activator*-nimistä luokkaa.

Jatketaan eteenpäin ja luodaan abstrakti luokka nimeltä *CommProtocol*. Määrittelen luokan omassa DLL:ssään, jotta se voidaan jakaa useiden DLL:ien kanssa, jotka haluavat periytyä siitä. (Huomaa, että käskyrivin parametrit on upotettu koodin kommentteihin.)

```
// CommProtocol.cs
// Muodostettu seuraavalla käskyrivikomennolla:
//         csc /t:library commprotocol.cs
public abstract class CommProtocol
{
    public static string DLLMask = “CommProtocol*.dll";
    public abstract void DisplayName();
}
```

Luon nyt kaksi erillistä DLL:ää, jotka kumpikin kuvaavat tiedonsiirtoprotokollaa ja sisältävät luokan, joka on periytetty abstraktista luokasta *CommProtocol*. Huomaa, että molemmat tarvitsevat käännöksessä viittauksen CommProtocol.dll:ään. Tässä IGN DLL:

```
// CommProtocolIGN.cs
// Muodostettu seuraavalla käskyrivikomennolla:
//         csc /t:library CommProtocolIGN.cs /r:CommProtocol.dll
using System;

public class CommProtocolIGN : CommProtocol
{
    public override void DisplayName()
```

(jatkuu)

```
    {
        Console.WriteLine("This is the IBM Global Network");
    }
}
```

Ja tässä TCP/IP DLL:

```
// CommProtocolTcpIp.cs
// Muodostettu seuraavalla käskyrivikomennolla:
//         csc /t:library CommProtocolTcpIp.cs /r:CommProtocol.dll
using System;

public class CommProtocolTcpIp : CommProtocol
{
    public override void DisplayName()
    {
        Console.WriteLine("This is the TCP/IP protocol");
    }
}
```

Katsotaan nyt, miten helppoa on dynaamisesti ladata kooste, selvittää tyyppi, instantioida tuota tyyppiä oleva objekti ja kutsua yhtä sen metodia (Kirjan mukana tulevalla cd:llä on muuten BuildLateBinding.cmd-niminen komentotiedosto, joka muodostaa nämä tiedostot.)

```
using System;
using System.Reflection;
using System.IO;

class LateBindingApp
{
    public static void Main()
    {

        string[] fileNames = Directory.GetFiles
                             (Environment.CurrentDirectory,
                              CommProtocol.DLLMask);
        foreach(string fileName in fileNames)
        {
            Console.WriteLine("Loading DLL '{0}'", fileName);

            Assembly a = Assembly.LoadFrom(fileName);

            Type[] types = a.GetTypes();
            foreach(Type t in types)
            {
                if (t.IsSubclassOf(typeof(CommProtocol)))
```

```
                {
                    object o = Activator.CreateInstance(t);

                    MethodInfo mi = t.GetMethod("DisplayName");

                    Console.Write("\t");
                    mi.Invoke(o, null);
                }
                else
                {
                    Console.WriteLine("\tThis DLL does not have " +
                    "CommProtocol-derived class defined");
                }
            }
        }
    }
}
```

Käytin ensin *System.IO.Directory*-luokkaa hakiessani kansiosta kaikki hakuehtoon *CommProtocol*.dll* täsmäävät DLL:t. *Directory.GetFiles*-metodi palauttaa taulukon, joka sisältää ehtoon sopivat tiedostonimet. Sen jälkeen voin käyttää *foreach*-silmukkaa taulukon läpikäyntiin ja kutsua jokaisen kohdalla jo aiemmin tässä luvussa käytettyä *Assembly.LoadFrom*-metodia. Kun kooste on ladattu annetusta DLL:stä, käyn läpi sen kaikki tyypit ja kutsun *Type.SubClassOf*-metodia määritelläkseni onko koosteessa tyyppiä, joka periytyy *CommProtocol*-luokasta. Oletan, että jos sellainen löytyy, niin DLL on kelvollinen. Kun löydän koosteen, joka sisältää *CommProtocol*-luokasta periytyvän tyypin, instantioin *Activator*-objektin ja välitän sen muodostimelle *type*-objektin. Kuten varmaan arvaatkin jo nimestä, *Activator*-luokkaa käytetään luomaan (tai aktivoimaan) tyyppi dynaamisesti.

Sen jälkeen käytän *Type.GetMethod*-metodia *MethodInfo*-objektin luomiseen määritellen metodin nimeksi *DisplayName*. Kun olen tehnyt sen, voin käyttää *MethodInfo*-objektin *Invoke*-metodia, välittäen sille parametrina aktivoidun tyypin ja - DLL:n *DisplayName*-metodia kusutaan!

Koodin luominen ja ajaminen suorituksen aikana

Ny t kun olet nähnyt, miten selvität tyypin tietoja suorituksen aikana, teet myöhäisen sidonnan ja suoritat koodia dynaamisesti, otetaan seuraava looginen askel ja luodaan koodia "lennossa". Tyyppien luominen suorituksen aikana tapahtuu käyttäen *System.Reflection.Emit*-nimiavaruutta. Nimiavaruuden luokkien avulla voit määritellä koosteen, luoda modulin koosteeseen, määritellä uusia tyyppejä moduliin (mukaanlukien sen jäsenet) ja vieläpä lisätä MSIL-käskyjä sovelluksen logiikkaa varten.

Vaikka tämän esimerkin koodi on äärimmäisen yksinkertainen, olen erottanut palvelinkoodin (DLL:n, joka sisältää luokan, joka luo *HelloWorld*-nimisen metodin) asiakaskoodista eli sovelluksesta, joka instantioi koodin generoivan luokan ja kutsuu sen *HelloWorld*-metodia. (Huomaa, että kääntäjän valitsimet on kerrottu kommenteissa.) Selitykset tulevat seuraavan DLL-koodin jälkeen:

```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace ILGenServer
{
    public class CodeGenerator
    {
        public CodeGenerator()
        {
            // Haetaan currentDomain.
            currentDomain = AppDomain.CurrentDomain;

            // Luodaan kooste.
            assemblyName = new AssemblyName();
            assemblyName.Name = "TempAssembly";

            assemblyBuilder =
                    currentDomain.DefineDynamicAssembly
                    (assemblyName, AssemblyBuilderAccess.Run);

            // luodaan moduli koosteeseen
            moduleBuilder = assemblyBuilder.DefineDynamicModule
                                ("TempModule");

            // luodaan tyyppi moduliin
            typeBuilder = moduleBuilder.DefineType
                              ("TempClass",
                              TypeAttributes.Public);
            // lisätään jäsen (metodi) tyyppiin
            methodBuilder = typeBuilder.DefineMethod
                                ("HelloWorld",
                                 MethodAttributes.Public,
                                 null,null);
```

```
            // Generoidaan MSIL.

            msil = methodBuilder.GetILGenerator();
            msil.EmitWriteLine("Hello World");
            msil.Emit(OpCodes.Ret);

            // Viimeinen askel : luodaan tyyppi.
            t = typeBuilder.CreateType();
        }

        AppDomain currentDomain;
        AssemblyName assemblyName;
        AssemblyBuilder assemblyBuilder;
        ModuleBuilder moduleBuilder;
        TypeBuilder typeBuilder;
        MethodBuilder methodBuilder;
        ILGenerator msil;
        object o;

        Type t;
        public Type T
        {
            get
            {
                return this.t;
            }
        }
    }
}
```

Aluksi instantioimme *AppDomain*-objektin sovelluksen domainissa. (Näet luvussa 17, "Yhteistoiminta hallitsemattoman koodin kanssa," että sovelluksen domainit ovat samanlaisia kuin Win32:n prosessit.) Sen jälkeen instantioimme *AssemblyName*-objektin. *AssemblyName*-luokasta puhutaan yksityiskohtaisesti luvussa 18, mutta voin tässä sanoa, että se on luokka, jota koostevaraston ylläpito-ohjelma käyttää hakiessaan tietoja koosteesta. Kun meillä on sovelluksen domain ja alustettu koosteen nimi, luomme uuden koosteen *AppDomain.DefineDynamicAssembly*-metodilla. Huomaa, että sille välitettävät parametrit ovat koosteen nimi sekä muoto, jolla koostetta voidaan käsitellä. *AssemblyBuilderAccess.Run*-muoto tarkoittaa, että kooste voidaan suorittaa muistissa, mutta sitä ei voi tallentaa. *AppDomain.DefineDynamicAssembly*-metodi palauttaa *AssemblyBuilder*-objektin, jonka muunnamme *Assembly*-objektiksi. Tässä vaiheessa meillä on muistissa täysin toimiva kooste. Nyt meidän pitää luoda sen tilapäinen moduli ja moduliin tyyppi.

Aloitamme kutsumalla *Assembly.DefineDynamicModule*-metodia saadaksemme *ModuleBuilder*-objektin. Kun meillä on se, kutsumme sen *DefineType*-metodia luodaksemme *TypeBuilder*-objektin, välittäen sille paramerina tyypin nimen ("*TempClass*") ja attribuutit,

jotka määrittelevät sen (*TypeAttribute.Public*). Nyt kun meillä on käsissämme *TypeBuilder*-objekti, voimme luoda minkä tyypin haluamme. Tässä tapauksessa luomme metodin käyttämällä *TypeBuilder.DefineMethod*-metodia.

Lopulta meillä on uusi *TempClass*-niminen tyyppi, jossa on metodi nimeltä *HelloWorld*. Nyt meidän pitää päättää, mitä koodia tähän metodiin sijoitetaan. Koodi tehdään *ILGenerator*-objektilla käyttäen *MethodBuilder.GetILGenerator*-metodia ja kutsuen erilaisia *ILGenerator*-metodeja koodin kirjoittamiseen metodiin.

Huomaa, että voimme tässä käyttää tavallista koodia, kuten *Console.WriteLine*, käyttämällä erilaisia ILGenerator-metodeja tai voimme tehdä MSIL-käskyjä käyttämällä *ILGenerator.Emit*-metodia. *ILGenerator.Emit*-metodi ottaa ainoana parametrinaan *OpCodes*-luokan jäsenkentän, joka on liitetty suoraan MSIL-koodiin.

Lopuksi kutsumme *TypeBuilder.CreateType*-metodia. Tämän tulee aina olla viimeinen suoritettava vaihe sen jälkeen, kun olet määritellyt uuden tyypin jäsenet. Sen jälkeen haemme uuden tyypin *Type*-objektin käyttämällä *Type.GetType*-metodia. Tämä objekti tallennetaan jäsenmuuttujaan, josta asiakasohjelma voi sen myöhemmin hakea.

Nyt asiakasohjelman täytyy vain hakea *CodeGenerator*-luokan *Type*-jäsen, luoda *Activator*-luokan instanssi, instantioida tyypistä *MethodInfo*-objekti ja sen jälkeen käynnistää metodi. Tässä koodi joka tekee sen, lisättynä virhetarkistuksella, joka varmistaa, että kaikki toimii niin kuin pitää:

```
using System;
using System.Reflection;
using ILGenServer;

public class ILGenClientApp
{
    public static void Main()
    {
        Console.WriteLine(“Calling DLL function to generate “ +
                          "a new type and method in memory...”);
        CodeGenerator gen = new CodeGenerator();

        Console.WriteLine(“Retrieving dynamically generated type...”);
        Type t = gen.T;
        if (null != t)
        {
            Console.WriteLine(“Instantiating the new type...”);
            object o = Activator.CreateInstance(t);
```

```
            Console.WriteLine("Retrieving the type's " +
                              "HelloWorld method...");
            MethodInfo helloWorld = t.GetMethod("HelloWorld");
            if (null != helloWorld)
            {
                Console.WriteLine("Invoking our dynamically " +
                                  "created HelloWorld method...");
                                  helloWorld.Invoke(o, null);
            }
            else
            {
                Console.WriteLine("Could not locate " +
                                  "HelloWorld method");
            }
        }
        else
        {
            Console.WriteLine("Could not access Type from server");
        }
    }
}
```

Jos nyt käännät ja käynnistät tämän sovelluksen, näet seuraavat tulokset:

```
Calling DLL function to generate a new type and method in memory...
Retrieving dynamically generated type...
Instantiating the new type...
Retrieving the type's HelloWorld method...
Invoking our dynamically created HelloWorld method...
Hello World
```

Yhteenveto

Reflection-menetelmällä voidaan selvittää tyypin tietoja suorituksen aikana. Reflection-APIn avulla voit esimerkiksi käydä läpi koosteen modulit ja koosteen tyypit ja hakea tyypin erilaisia ominaisuuksia. Kehittyneempiä reflection-menetelmän mahdollisuuksia ovat esimerkiksi mahdollisuus käynnistää metodeja dynaamisesti, käyttää tyyppejä (myöhäisellä sidonnalla) ja jopa luoda ja suorittaa MSIL-koodia suorituksen aikana.