17

Yhteistoiminta hallitsemattoman koodin kanssa

Uusi kieli tai kehitysympäristö ei tule olemaan pitkäikäinen, jos se jättää huomioimatta vanhat järjestelmät ja koodit ja antaa mahdollisuuden vain uusien sovellusten kirjoittamiseen. Riippumatta siitä, kuinka hieno uusi järjestelmä on, sen tekijöiden tulee ottaa huomioon aika, jolloin uutta järjestelmää käytetään yhdessä vanhan kanssa. Tämän huomioiden .NET ja C#-kehitystiimit ovat tehneet ohjelmoijalle helpoksi käyttää olemassaolevaa hallitsematonta koodia. *Hallitsematon koodi* (unmanaged code) tarkoittaa koodia, joka ei ole hallittu eli .NET ajonaikaisen ympäristön ohjauksessa. Tässä luvussa käsittelen kolme pääesimerkkiä hallitsemattomasta koodista .NETissä:

- **Platform Invocation -palvelut** Näiden palvelujen avulla .NET-koodi voi käsitellä olemassaolevissa hallitsemattomissa DLL:ssä olevia funktioita, tietueita ja jopa takaisinkutsuja.
- **Turvaton koodi** Kirjoittamalla turvatonta koodia (unsafe code) C#-ohjelmoija voi käyttää rakenteita (kuten osoittimia) C#-sovelluksissa sen kustannuksella, että koodia ei hallitse .NETin ajonaikainen ympäristö.
- **COM-yhteistoiminta** Tämä termi tarkoittaa .NET-koodin kykyä käyttää COM-komponentteja ja COM-sovellusten mahdollisuuteen käyttää .NET-komponentteja.

Platform Invocation -palvelut

.NET Platform Invocation -palvelut (joita kutsutaan joskus nimellä *PInvoke*) mahdollistavat hallitun koodin käyttää funktioita ja tietueita, jotka on haettu DLL:stä. Tässä kappaleessa kerron, miten kutsutaan DLL-funktioita sekä niitä attribuutteja, joita käytetään muotoilemaan (marshal) tiedot .NET-sovelluksen ja DLL:n välillä.

Koska sinulla ei ole tarjota DLL-funktion lähdekoodia C#-kääntäjälle, sinun pitää määritellä kääntäjälle metodin otsikkorivi sekä antaa tiedot paluuarvoista ja siitä, miten parametrit muotoillaan DLL:lle.

> Huomaa Kuten tiedät, voit tehdä DLL:iä C#:lla ja muilla .NET-kääntäjillä. Tässä luvussa vältän käyttämästä termiä "hallitsematon Win32 DLL." Voit olettaa, että aina kun puhun DLL:stä, tarkoitan sen hallitsematonta versiota.

Käytettävän DLL-funktion määritteleminen

Ensimmäisenä asiana katsomme, miten määritellään yksinkertainen DLL-funktio C#:ssa. Aloitamme jo perinteiseksi muodostuneella .NET *PInvoke*-esimerkillä, Win32 "MessageBox"-funktiolla. Sen jälkeen siirrymme kehittyneempiin parametrien muotoilutapoihin.

Kuten opit luvussa 8, "Attribuutit," attribuutit sisältävät suunnittelun aikana annettua informaatiota C#-tyypeistä. Reflection-menetelmän avulla tätä informaatiota voidaan myöhemmin suorituksen aikana kysellä. C# käyttää attribuuttia antaessaan sinun määritellä kääntäjälle sen DLL-funktion, jota aiot kutsua. Attribuutti on nimeltään *DllImport* ja sen syntaksi on seuraava:

[DllImport(*dllNimi*)]
käsittelymääre static extern *paluuarvo dllFunktio*(*parametri1*, *parametri2*,...);

Kuten näet, DLL-funktion mukaan ottaminen onnistuu yksinkertaisesti liittämällä *DllImport*-attribuutti (välittämällä DLL:n nimen sen muodostimelle) siihen DLL-funktioon, jota haluat kutsua. Huomaa, että sinun pitää käyttää määriteltävässä funktiossa myös *static* ja *extern*-määreitä. Seuraava *MessageBox*-esimerkki näyttää, miten helppoa PInvoke-metodia on käyttää:

```
using System;
using System.Runtime.InteropServices;
```

```
class PInvoke1App
{
    [DllImport("user32.dll")]
    static extern int MessageBoxA(int hWnd,
                                  string msg,
                                  string caption,
                                  int type);

    public static void Main()
    {
        MessageBoxA(0,
                    "Hello, World!",
                    "This is called from a C# app!",
                    0);
    }
}
```

Sovelluksen ajaminen aiheuttaa odotetun viesti-ikkunan avautuvan ja tulostavan "Hello World"-tekstin.

Huomaa *using*-määre, joka viittaa *System.Runtime.InteropServices*-nimiavaruuteen. Se on nimiavaruus, joka määrittelee *DllImport*-attribuutin. Huomaa sen jälkeen, että olen määritellyt *MessageBoxA*-metodin, jota kutsutaan *Main*-metodissa. Mutta entä jos haluat kutsua sisäistä C#-metodiasi jollain muulla kuin DLL-funktion nimellä? Voit tehdä sen käyttämällä yhtä *DllImport*-atribuutin nimettyä parametria.

Seuraavassa koodissa kerron kääntäjälle, että haluan metodiani kutsuttavan nimellä MessageBoxA. Koska en määritellyt DLL-funktion nimeä *DllImport*-attribuutissa, kääntäjä olettaa, että molemmat nimet ovat samoja.

```
    [DllImport("user32.dll")]
    static extern int MessageBoxA(int hWnd,
                                  string msg,
                                  string caption,
                                  int type);
```

Jotta näkisemme, miten muutos vaikuttaa oletuskäyttäytymiseen, katsotaan toista esimerkkiä. Tällä kertaa käytän sisäistä nimeä *MsgBox,* vaikka ohjelma yhä kutsuu DLL:n *MessageBoxA*-funktiota.

```
using System;
using System.Runtime.InteropServices;

class PInvoke2App
```

(jatkuu)

```
{
    [DllImport("user32.dll", EntryPoint="MessageBoxA")]
    static extern int MsgBox(int hWnd,
                             string msg,
                             string caption,
                             int type);

    public static void Main()
    {
        MsgBox(0,
               "Hello, World!",
               "This is called from a C# app!",
               0);
    }
}
```

Kuten näet, minun tarvitsee vain määritellä *DllImport*-attribuutin nimetty parametri *EntryPoint* voidakseni nimetä ulkoisen DLL:n funktion sisäisessä käytössä miten haluan.

Viimeinen seikka ennen siirtymistä parametrien muotoiluum, on *CharSet*-parametri. Parametrin avulla voit määritellä DLL-tiedoston käyttämän merkistön. Kun kirjoitetaan C++-sovelluksia, ei normaalisti tarvitse eksplisiittisesti määritellä *MessageBoxA* tai *MessageBoxW*-funkitota, sillä *pragma* on jo kertonut kääntäjälle käytätkö ANSI vai UNICODE-merkistöä. Sen vuoksi kutsut C++:ssa *MessageBox*-funktiota ja kääntäjä määrittelee, kumpaa merkistöä käytät. Vastaavasti C#:ssa voit määritellä kohteen merkistön *DllImport*-attribuutille, jolloin kohteen merkistö osoittaa, mitä versioita *MessageBox*-funktiosta kutsutaan. Seuraavassa esimerkissä kutsutaan yhä *MessageBoxA*-funktiota sen arvon takia, jonka välitän *DllImport*-attribuutille sen *CharSet*-nimetun parametrin kautta.

```
using System;
using System.Runtime.InteropServices;

class PInvoke3App
{
    // CharSet.Ansi will result in a call to MessageBoxA.
    // CharSet.Unicode will result in a call to MessageBoxW.
    [DllImport("user32.dll", CharSet=CharSet.Ansi)]
    static extern int MessageBox(int hWnd,
                                 string msg,
                                 string caption,
                                 int type);

    public static void Main()
```

```
    {
        MessageBox(0,
                   "Hello, World!",
                   "This is called from a C# app!",
                   0);
    }
}
```

Etu *CharSet*-parametrin käytöstä on se, että voin asettaa muuttujan sovelluksessani ja antaa sen määrätä, kumpaa versiota (ANSI vai UNICODE) funktiosta kutsutaan. Minun ei tarvitse muuttaa kaikkea koodia, jos päätän vaihtaa versiosta toiseen.

Takaisinkutsufunktioiden käyttäminen C#:ssa

C#-sovellukset eivät pelkästään voi kutsua DLL-funktioita, vaan myös DLL-funktiot voivat kutsua sovelluksesi määrättyjä C#-metodeja takaisinkutsutilanteissa. Takaisinkutsut käsittävät minkä tahansa Win32 EnumXXX-funktion käytön, jossa kutsut funktiota luetellaksesi jotain, välittäen sille funktio-osoittimen, jota Windows-kutsuu jokaista löydettyä luettelon jäsentä kohden. Tämä toteutetaan *PInvoke*-metodin (kutsu DLL-funktioon) ja delegaatin (takaisinkutsun määritteleminen) yhteistyöllä. Jos sinun pitää päivittää tietosi delegaateista, käy uudelleen läpi luku 14, "Delegaatit ja tapahtuma-käsittelijät."

Seuraava koodi luetteloi ja tulostaa otsikkorivin tekstin kaikista järjestelmän ikkunoista:

```
using System;
using System.Runtime.InteropServices;
using System.Text;

class CallbackApp
{
 [DllImport("user32.dll")]
 static extern int GetWindowText(int hWnd, StringBuilder text, int
 count);

 delegate bool CallbackDef(int hWnd, int lParam);

 [DllImport("user32.dll")]
 static extern int EnumWindows (CallbackDef callback, int lParam);

 static bool PrintWindow(int hWnd, int lParam)
    {
        StringBuilder text = new StringBuilder(255);
```

(jatkuu)

```
        GetWindowText(hWnd, text, 255);

        Console.WriteLine("Window caption: {0}", text);
        return true;
    }

    static void Main()
    {
        CallbackDef callback = new CallbackDef(PrintWindow);
        EnumWindows(callback, 0);
    }
}
```

Aluksi määrittelen Win32-funktiot *EnumWindows* ja *GetWindowText* käyttämällä *DllImport*-attribuuttia. Sitten määrittelen *CallbackDef*-nimisen delegaatin ja metodin nimeltä *PrintWindows*. Sen jälkeen minun pitää enää instantioida *Main*-metodissa *CallbackDef*-delegaatti (välittäen sille parametrina *PrintWindows*-metodin) ja kutsua *EnumWindows*-metodia. Kunkin järjestelmästä löytyneen ikkunan kohdalla Windows kutsuu *PrintWindows*-metodia.

PrintWindows-metodi on mielenkiintoinen, koska se käyttää *StringBuilder*-luokkaa luodakseen kiinteämittaisen merkkijonon, joka välitetään *GetWindowText*-funktiolle. Siksi *GetWindowText*-funktio määritellään seuraavasti:

```
static extern int GetWindowText(int hWnd, StringBuilder text, int count);
```

Syy tähän kaikkeen on se, että DLL-funktion ei sallita muuttavan merkkijonoa, joten et voi käyttää sitä tyyppiä. Ja vaikka yrittäisit välittää viittauksen, ei kutsuvalla koodilla ole mitään tapaa alustaa merkkijonoa oikean kokoiseksi. Tässä kohtaa tarvitaan *StringBuilder*-luokkaa. Kutsuttu funkito voi viitata *StringBuilder*-objektiin ja muokata sitä, kunhan tekstin pituus ei ylitä maksimiarvoa, joka on välitetty *StringBuilder*-luokan muodostimelle.

Muotoilu ja *PInvoke*

Vaikka et yleensä näe muotoilua (marshaling) tai määrittele, miten se toimii, joka kerta, kun kutsut DLL-funktiota, .NETin pitää muotoilla funktiolle menevät parametrit ja kutsuvalle.NET-sovelluksella takaisin tulevat paluuarvot. Minun ei tarvinnut tämän luvun edeltävissä esimerkeissä tehdä mitään saadakseni muotoilun tapahtumaan, koska .NET on määritellyt oletustyypin kullekin .NETin tyypille. Esimerkiksi *MessageBoxA* ja *MessageBoxW* -funktioiden toinen ja kolmas parametri oli määritelty string-tyypiksi. C#-kääntäjä kuitenkin tietää, että C#:n string-tyyppiä vastaava on Win32 LPSTR. Mutta mitä tapahtuu, jos haluat korvata .NETin muotoilun oletukset? Voit tehdä sen käyttämällä *MarshalAs*-attribuuttia, joka on myös määritelty *System.Runtime.InteropServices*-nimiavaruudessa.

Käytän seuraavassa esimerkissä jälleen *MessageBox*-funktiota pitääkseni asiat yksinkertaisina. Olen päättänyt käyttää nyt Win32 *MessageBox*-funktion Unicode-versiota. Kuten tiedät edellisistä kappaleista, minun pitää vain välittää lueteltu tyyppi *CharSet.Unicode DllImport*-attribuutin nimetylle parametrille *CharSet*. Tällä kertaa haluan kuitenkin kääntäjän muotoilevan tiedon LPWSTR-tyyppiseksi, joten käytän *MarshalAs*-attribuuttia ja määrittelen luetellun tyypin *UnmanagedType* avulla tyypin, johon haluan muotoilun tehdä. Tässä koodi:

```
using System;
using System.Runtime.InteropServices;

class PInvoke4App
{
    [DllImport("user32.dll", CharSet=CharSet.Unicode)]
    static extern int MessageBox(int hWnd,
                                 [MarshalAs(UnmanagedType.LPWStr)]
                                 string msg,
                                 [MarshalAs(UnmanagedType.LPWStr)]
                                 string caption,
                                 int type);

    public static void Main()
    {
        MessageBox(0,
                   "Hello, World!",
                   "This is called from a C# app!",
                   0);
    }
}
```

Huomaa, että *MarshalAs*-attribuutti voidaan liittää metodin parametreihin (kuten tässä esimerkissä), metodin paluuarvoihin ja luokkien ja tietueiden kenttiin. Huomaa myös, että jos haluat muuttaa oletusmuotoilua metodin paluuarvolle, sinun tulee liittää *MarshalAs* -attribuutti itse metodiin.

Turvattoman koodin kirjoittaminen

Yksi seikka, joka monia pohdituttaa siirryttäessä C++:sta C#:iin, liittyy siihen, voiko C#:ssa hallita täysin muistia tapauksissa, joissa se on tarpeen. Tämä asia liittyy turvattomaan koodiin. Huolimatta pahaenteiseltä kuulostavalta nimestään, *turvaton koodi* (unsafe code) ei luonnostaan ole turvatonta ja epäluotettavaa. Se on koodi, jossa .NETin ajonaikainen ympäristö ei suorita muistin varaamista ja vapauttamista. Mahdollisuus kirjoittaa turvatonta koodia on parhaimmillaan, kun käytät osoittimia vanhan koodin kanssa (kuten C:n APIn)

tai kun sovelluksesi tarvitsee muistin suoraa käsittelyä (yleensä suorituskyvyn parantamiseksi).

Kirjoitat turvatonta koodia käyttämällä kahta avainsanaa: *unsafe* ja *fixed*. *unsafe*-avainsana määrittää, että merkitty lohko suoritetaan turvattomassa ympäristössä. Sitä voidaan soveltaa kaikkiin metodeihin, mukaanlukien muodostimet ja ominaisuudet, ja jopa metodin sisällä oleviin koodilohkoihin. *fixed*-avainsana on vastuussa hallitun objektin kiinnittämisestä. *Kiinnittäminen* (pinning) on toiminto, joka ilmoittaa roskienkeruulle (GC), että kyseistä objektia ei voi siirtää. Sovelluksen suorituksen aikana objekteille varataan tilaa ja sitä vapautetaan, jolloin muistiin jää "aukkoja". .NETin ajonaikainen ympäristö ei kuitenkaan anna muistin pirstoutua vaan siirtää objekteja, jotta muistia käytetään tehokkaammin. Ei tietenkään ole kivaa, jos sinulla on osoitin määrättyyn muistialueeseen ja tietämättäsi .NETin ajonaikainen ympäristö siirtää objektin muualle, jättäen sinulle viallisen osoitteen. Koska GC siirtää objekteja muistissa sovelluksen tehokkuuden parantamiseksi, sinun tulee käyttää *fixed*-avainsanaa harkiten.

Osoittimien käyttäminen C#:ssa

Katsotaan ensin muutamia sääntöjä, jotka liittyvät osoittimien ja turvattoman koodin käyttämiseen ja siirrytään sen jälkeen esimerkkeihin. Osoittimia voidaan hankkia vain arvotyyppeihin, taulukoihin ja merkkijonoihin. Kun kyseessä on taulukko, pitää ensimmäisen elementin olla arvotyyppi, koska C# palauttaa tosiasiallisesti osoittimen taulukon ensimmäiseen elementtiin eikä itse taulukkoon. Siksi kääntäjän kannalta se palauttaa edelleen osoittimen arvotyyppiin eikä viittaustyyppiin.

Taulukossa 17-1 kerrotaan, miten tyypillisiä C/C++:n osoittimiin liittyviä käsitteitä ylläpidetään C#:ssa:

Taulukko 17-1 C/C++ Osoitinoperaattorit

Operaattori	Kuvaus
&	*address-of*-operaattori palauttaa osoittimen, joka sisältää muuttujan muistiosoitteen.
*	*dereference*-operaattoria käytetään osoittimen osoittaman arvon selvittämiseen.
->	*dereferencing and member access* -operaattoria käytetään jäsenen ja osoittimen käsittelyyn.

Seuraava esimerkki näyttää tutulta jokaiselle C ja C++ -ohjelmoijalle. Kutsun siinä metodia, joka saa parametrina kaksi osoitinta *int*-tyyppisiin muuttujiin. Metodi muuttaa niiden arvoa ennen kuin palauttaa kutsujalle. Ei erityisen jännittävää, mutta näyttää, miten osoittimia käytetään C#:ssa.

```
// Käännä tämä sovellus /unsafe valitsimella.

using System;

class Unsafe1App
{
    public static unsafe void GetValues(int* x, int* y)
    {
        *x = 6;
        *y = 42;
    }

    public static unsafe void Main()
    {
        int a = 1;
        int b = 2;
        Console.WriteLine("Before GetValues() : a = {0}, b = {1}",
                          a, b);
        GetValues(&a, &b);
        Console.WriteLine("After GetValues() : a = {0}, b = {1}",
                          a, b);
    }
}
```

Tämä esimerkki pitää kääntää /unsafe -valitsimella. Sovelluksen tuloste näyttää seuraavalta:

```
Before GetValues() : a = 1, b = 2
After GetValues() : a = 6, b = 42
```

fixed-käsky

fixed-käskyn syntaksi on seuraava:

fixed (*type* ptr = lauseke*) *käsky*

Kuten mainitsin, käsky kertoo GC:lle, että sen ei kannata vaivautua tämän muuttujan takia. Huomioi, että *type* on hallitsematon tyyppi tai *void*, *lauseke* on mikä tahansa lauseke, jonka tulos on *type*-osoitin ja *käsky* viittaa koodilohkoon, johon muuttujan kiinnittämistä on sovellettu. Seuraavalla sivulla on yksinkertainen esimerkki.

```
using System;

class Foo
{
    public int x;
}

class Fixed1App
{
    unsafe static void SetFooValue(int* x)
    {
        Console.WriteLine(“Dereferenced pointer to modify foo.x”);
        *x = 42;
    }

    unsafe static void Main()
    {
        // Luodaan luokan instanssi.
        Console.WriteLine(“Creating the Foo class”);
        Foo foo = new Foo();

        Console.WriteLine(“foo.x intialized to {0}", foo.x);

        // fixed-käsky kiinnittää foo-objektin kunnes
        // koostekäsky päättyy sulkuun.
        Console.WriteLine(“Setting pointer to foo.x”);
        // Sijoitetaan foo-objektin osoite muuttujaan Foo*.
        fixed(int* f = &foo.x)
        {
            Console.WriteLine(“Calling SetFooValue passing “ +
                              “pointer to foo.x”);
            SetFooValue(f);
        }

        // Näytetään, että muutimme jäsentä osoittimella.
        Console.WriteLine(“After return from “ +
                          “SetFooValue, foo.x = {0}", foo.x);
    }
}
```

Koodi instantioi *Foo*-luokan ja *fixed*-käskyllä kiinnittää objektin ja hakee sen ensimmäisen jäsenen osoitteen *int**-tyyppiseen muuttujaan (*SetFooValue*-metodin tarvitsema tyyppi). Huomaa, että *fixed*-käskyä käytetään vain siinä koodin osassa, jossa *Foo*-objektin siirtämisellä olisi vaikutusta. Tämä on pieni mutta tärkeä seikka isommissa koodilohkoissa, joissa tulee minimoida aika, jonka objekti on kiinnitettynä. Ylläolevan koodin kääntäminen ja suorittaminen aiheuttaa seuraavan tulosteen:

```
Creating the Foo class
foo.x intialized to 0
Setting pointer to foo.x
Calling SetFooValue passing pointer to foo.x
Dereferenced pointer to modify foo.x
After return from SetFooValue, foo.x = 42
```

> Huomaa Tärkeä kiinnitettyihin muuttujiin liittyvä seikka on se, että C#-kääntäjä ei rajoita kiinnitetyn muuttujan käsittelyä pelkästään turvattomaksi merkitylle koodialueelle. Voit esimerkiksi käyttää kiinnitettyä muuttujaa sijoituslauseen oikealla puolella ja sijoitaa sen arvon muuttujaan, joka on määritelty laajemmalla näkyvyysalueella kuin turvaton lohko. Täten voit käyttää turvatonta arvoa turvattoman lohkon ulkopuolella. Kääntäjä ei anna tästä varoitusta ja onkin ohjelmoijan vastuulla olla huolellinen, kun kiinnitettyä muuttujaa käytetään sijoituslauseen oikealla puolella.

Yhteistyö COMin kanssa

Jos olet miettinyt, miten kaikki nuo COM-komponentit, joita olet vuosien aikana kirjoittanut, toimivat .NETin ajonaikaisessa ympäristössä, on tämä kappale sinua varten. Näytän sinulle, miten perinteiset COM-komponentit (yäk, minua inhottaa nähdä COMia kutsuttavan perinteiseksi COMiksi) sijoittuvat .NET-maailmaan.

Uljas uusi maailma

Kuten olet nähnyt tässä kirjassa, ei ole epäilystäkään, etteivät .NET-ympäristö ja C#-kieli muodosta tehokasta mallia komponenttipohjaisen järjestelmän rakentamiseen. Mutta mitä niistä tuhansista olemassa olevista uudelleenkäytettävistä COM-komponenteista, joita olet tehnyt viime vuosien aikana puhumattakaan niiden tekoon kuluneista kahvista ja unettomista öistä? Toimivatko ne käsi kädessä .NETin hallitun ajoympäristön kanssa? Meille, jotka teemme COM-ohjelmia työksemme ja niille, jotka elävät "COM on hyvä"-maailmassa, saavat kuulla nyt hyviä uutisia. COM on täällä jäädäkseen ja .NETin hallitut sovellukset voivat hyödyntää olemassa olevia COM-komponentteja. Kuten tulet näkemään, perinteiset COM-komponentit toimivat .NETin ajonaikaisen ympäristön kanssa yhteistyökerroksen (COM interop) kautta, joka käsittelee kaikki viestit, jotka hallitun ympäristön ja COM-komponentin hallitsemattoman valtakunnan välillä kulkevat.

Perusta

Koska koko COM Interop -kerros voi olla turhan massiivinen heti nieltäväksi, jätetään kaikki tekniset määrittelyt hetkeksi ja hypätään todelliseen esimerkkiin, jossa käytetään COM-komponenttia .NET-sovelluksesta. Kun jatkamme, kerron mitä tapahtuu ja miten pystyt käyttämään oppimaasi omissa sovelluksissasi.

Tässä esimerkissä oletamme, että minulla on Microsoft Visual C++:lla ja ATL:llä kirjoitettu AirlineInfo-COM-komponentti. En aio käydä läpi kaikkia vaadittavia vaiheita tämän komponentin tekemiseksi, koska haluan keskittyä .NETin ja C#:in näkökulmaan. Selitän kuitenkin keskeisen koodin. Koko Visual C++-projekti löytyy kirjan mukana tulevalta cd:ltä.

COM-komponenttimme on suunniteltu palauttamaan määrätyn lentolinjan tiedot. Yksinkertaisuuden vuoksi komponentti palauttaa linjan Air Scooby IC 5678 tiedot ja muiden linjojen kohdalla se palauttaa virheen. Lisäsin tämän virhemekanismin, jotta näet, miten COM-komponentin aiheuttama virhe voidaan ottaa kiinni kutsuvassa .NET-asiakassovelluksessa.

Tässä COM-komponentin IDL:

```
interface IAirlineInfo : IDispatch
{
    [id(1), helpstring("method GetAirlineTiming")]
        HRESULT GetAirlineTiming([in] BSTR bstrAirline, [out,retval]
BSTR* pBstrDetails);

    [propget, id(2), helpstring("property
        LocalTimeAtOrlando")] HRESULT
        LocalTimeAtOrlando([out, retval] BSTR
        *pVal);
};
```

Ei mitään liian ihmeellistä, edes aloitteleville COM-ohjelmoijille. Meillä on *IAirlineInfo*-niminen rajapinta, jossa on kaksi metodia: *GetAirlineTiming* ja *LocalTimeAtOrlando*. Katsotaan nyt *GetAirlineTiming*-metodin toteutusta:

```
STDMETHODIMP CAirlineInfo::GetAirlineTiming(BSTR
    bstrAirline, BSTR *pBstrDetails)
{
    _bstr_t bstrQueryAirline(bstrAirline);
    if(NULL == pBstrDetails) return E_POINTER;

    if(_bstr_t("Air Scooby IC 5678") ==
        bstrQueryAirline)
```

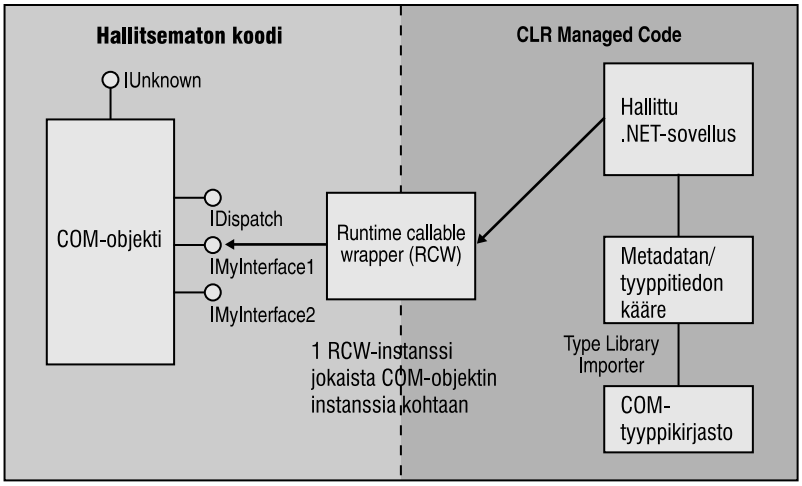
```
    {
        // Palautetaan tämän linjan aikataulutiedot.
        *pBstrDetails =
            _bstr_t(_T("16:45:00 - Will
            arrive at Terminal 3")).copy();
    }
        else
        {
            // Palautetaan virheilmoitus.
            return Error(LPCTSTR(_T("Not available" )),
                         __uuidof(AirlineInfo),
                         AIRLINE_NOT_FOUND);
    }
    return S_OK;
}
```

GetAirlineTiming-metodi ottaa kaksi parametria. Ensimmäinen (*bstrAirline*) on *BSTR,* joka esittää lentolinjaa ja toinen (*pBstrDetails*) on *lähtöparametri (*output parameter), joka palauttaa linjan tiedot (ajan ja portin). Metodissa tarkistamme, että *bstrAirline*-parametrin arvo on *"Air Scooby IC 5678"*. Jos on, palautamme kovakoodatut lennon tiedot. Jos parametrin arvo ei ole se, mitä odotimme, kutsumme paluuarvona virhemetodia osoittaen siten, että ohjelmamme tukee vain yhtä lentolinjaa.

Tällä perustietämyksellä komponentista katsomme seuraavaksi, miten generoidaan metadata komponentin tyyppikirjastosta, jotta .NET-asiakasohjelma voi käyttää metadataa keskustellessaan komponentin kanssa ja kutsuessaan sen metodeja.

Metadatan generointi COMin tyyppikirjastosta

.NET-sovellus, jonka pitää keskustella COM-komponenttimme kanssa, ei voi suoraan selvittää komponentin tarjoamia toimintoja. Miksi ei? Kuten näimme luvussa 16, "Metadatan kyseleminen Reflection-metodien avulla," .NETin ajonaikainen ympäristö on suunniteltu toimimaan sellaisten komponenttien kanssa, joilla on metadata, kun taas COM on suunniteltu toimimaan Rekisterin ja komponentin toteuttamien kyselymetodien avulla. Siksi ensimmäinen asia, joka meidän tulee tehdä, jotta .NET-maailma voisi käyttää COM-komponenttiamme, on generoida sille metadata. COM-komponentin tapauksessa tätä metadatakerrosta käyttää ajonaikainen ympäristö määritellessään tyypin tiedot. Näitä tyypin tietoja käytetään sitten suorituksen aikana tuottamaan *runtime callable wrapper (RCW).* (Katso kuva 17-1.) RCW suorittaa COM-objektin tosiasiallisen aktivoinnin sekä hoitaa muotoilun, kun .NET-sovellus käyttää sitä. RCW tekee myös paljon muuta, kuten hallitsee objektin identiteetin, objektin elinajan ja rajapinnan tallennuksen.



Kuva 17-1 .NETin ja COM-komponentin välisen yhteistoiminnan perusosat.

Objektin elinajan hallinta on oleellinen asia, koska .NETin GC siirtää objekteja paikasta toiseen ja hallitsee niitä automaattisesti, kun ne eivät ole enää käytössä. RCW palvelee tarkoitusta antamalla .NET-sovellukselle käsityksen, että se käyttää hallittua .NET-komponenttia ja se antaa hallitsemattomassa tilassa toimivalle COM-komponentille vaikutuksen, että sitä kutsuu perinteinen COM-asiakasohjelma. RCW:n luonti ja käyttäytyminen riippuu sitä, onko sinulla myöhäinen vai aikainen sidonta COM-objektiin. RCW tekee piilossa kovasti töitä ja muuttaa kaikki metodikutsut vastaaviksi hallitsemattomassa maailmassa olevan COM-komponentin vtable-kutsuiksi. Se toimii eräänlaisena hyvän tahdon lähettiläänä hallitun maailman ja hallitsemattoman *IUnknown*-maailman välillä.

Nyt riittää puhe! Generoidaan AirlineInfo COM-komponentillemme RCW. Tarvitsemme siihen apuohjelmaa nimeltä *tlbimp.exe* (Type Library Importer). Se tulee .NET SDK:n mukana. Se lukee COM-tyyppikirjaston ja generoi siitä metadata-kääreen, joka sisältää tyyppitiedot muodossa, jonka .NETin ajonaikainen ympäristö ymmärtää. Jotta voisit tehdä tämän, sinun tulee asentaa demosovellus kirjan mukana tulevalta cd:ltä ja etsiä AirlineInfo-komponentti.

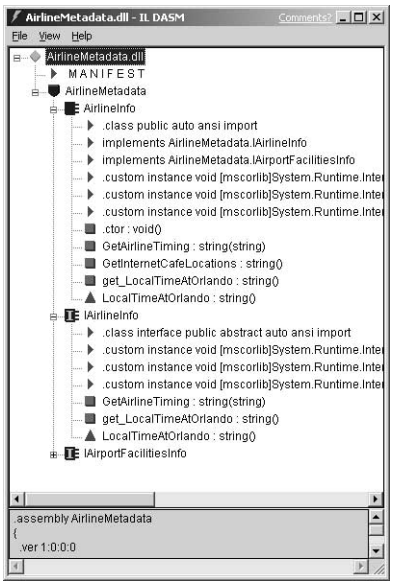
Kun olet tehnyt sen, kirjoita seuraava komentoriville:

```
TLBIMP AirlineInformation.tlb /out:AirlineMetadata.dll
```

Tämä pyytää *TLBIMP*-ohjelmaa lukemaan AirlineInfo-komponentin tyyppikirjaston ja generoimaan vastaavan metadata-kääreen nimeltä AirlineMetadata.dll. Jos kaikki menee kunnolla, näet seuraavan ilmoituksen:

```
TypeLib imported successfully to AirlineMetadata.dll
```

Minkälaista tietoa tyypistä tämä generoitu metadata sisältää ja miltä se näyttää? COM-ihmisinä olemme aina pitäneet suuressa arvossa rakastamaamme *OleView.exe*-apuohjelmaa, koska se muiden kykyjensä lisäksi antaa meidän vilkaista tyyppikirjaston sisältöä. Onneksi .NET SDK:ssa on jotain vastaavaa: *ILDASM* (ILDisassembler), joka esiteltiin luvussa 2, "Johdanto Microsoft .NETiin." Sen avulla voit tarkastella metadataa ja MSIL-koodia, joka on generoitu hallitulle koosteelle. Kuten opit luvussa 16, jokainen hallittu kooste sisältää itsensä kuvailevan metadatan ja ILDASM on erittäin käyttökelpoinen työkalu, kun haluat tarkastella sitä. Jatka eteenpäin ja avaa AirlineMetadata.dll ILDASMilla. Sinun pitäisi nähdä samanlainen ikkuna kuin kuvassa 17-2.



Kuva 17-2 ILDASM on erinomainen työkalu hallitun koosteen metadatan ja MSIL:n tutkimiseen.

Generoidusta metadatasta näet, että *GetAirlineTiming*-metodi on *AirlineInfo*-luokan julkinen jäsen. *AirlineInfo*-luokalle on myös generoitu muodostin. Huomaa, että metodin parametrit on automaattisesti vaihdettu vastaaviksi .NET-tyypeiksi. Tässä esimerkissä *BSTR* on korvattu tyypillä *System.String*. Huomaa myös, että parametri, joka on *GetAirlineTiming*-metodissa merkitty *[out,retval]*, on muunnettu metodin todelliseksi paluuarvoksi (palauttaa tyypin *System.String*). Lisäksi jokainen HRESULT-virhe, joka palautetaan COM-objektista (joko virheen tai virheellisen liiketoimintalogiikan vuoksi) aiheuttaa poikkeuksen.

Aikainen sidonta COM-komponentteihin

Nyt kun olemme generoineet metadatan, jota .NET-asiakasohjelma tarvitsee, yritetään käynnistää COM-objektimme *GetAirlineTiming*-metodi .NET-asiakasohjelmasta. Tässä C#-sovellus, joka luo COM-objektin käyttämällä aiemmin luomaamme metadataa ja kutsuu *GetAirlineTiming*-metodia. Huomaa, että tässä esimerkissä käytämme aikaista sidontaa. Aivan kohta näytän lisää esimerkkejä, joissa selvitetään tyyppi dynaamisesti ja käytetään myöhäistä sidontaa.

```
using System;
using System.Runtime.InteropServices;
using System.Reflection;
using AIRLINEINFORMATIONLib;

public class AirlineClient1App
{
    public static void Main()
    {
        ////////////////////////////////////////////
        /// AIKAINEN SIDONTA
        ////////////////////////////////////////////
        String strAirline = “Air Scooby IC 5678";
        String strFoodJunkieAirline = “Air Jughead TX 1234";
        try
        {
            AirlineInfo objAirlineInfo;

            // Luodaan uusi AirlineInfo-objekti.
            objAirlineInfo = new AirlineInfo();

            // Näytetään tuloste sen jälkeen,
            // kun on kutsuttu GetAirileTiming-metodia.
            Console.WriteLine(“Details for Airline {0} --> {1}",
            strAirline,objAirlineInfo.GetAirlineTiming(strAirline));

            // VIRHE: Seuraava aiheuttaa poikkeuksen!
            //
            // Console.WriteLine(“Details for Airline {0} --> {1}",
            // strFoodJunkieAirline,objAirlineInfo.GetAirlineTiming
            // (strFoodJunkieAirline));
        }
        catch(COMException e)
        {
            Console.WriteLine(“Oops- We encountered an error “ +
```

```
                                    "for Airline {0}. The Error message " +
                                    "is : {1}. The  Error code is {2}",
                                    strFoodJunkieAirline ,
                                    e.Message,e.ErrorCode);
        }
    }
}
```

Tässä ajonaikainen ympäristö muodostaa RCW:n, joka yhdistää metadata-luokan metodit ja kentät COM-objektin toteuttamiin rajapinnan metodeihin ja ominaisuuksiin. Kutakin COM-objektin instanssia kohden luodaan yksi RCW:n instanssi. .NETin ajonaikainen ympäristö huolehtii vain RCW:n elinkaaresta ja sen joutumisesta roskien keruuseen. RCW huolehtii sen COM-objektin viittauslaskennan ylläpidosta, johon se on liitetty, joten .NETin ajonaikaisen ympäristön ei tarvitse huolehtia siitä. Kuten näet kuvassa 17-2, AirlineInfo-metadata on määritelty nimiavaruudessa *AIRLINEINFORMATIONLib*. .NET-asiakasohjelma näkee kaikki rajapinnan metodit aivan kuin ne olisivat *AirlineInfo*-luokan jäseniä. Meidän pitää vain luoda instanssi *AirlineInfo*-luokasta *new*-operaattorilla ja kutsua luodun objektin *public*-tyyppisiä metodeja. Kun metodia kutsutaan, RCW muuntaa kutsun vastaavaksi COM-metodikutsuksi. RCW hoitaa myös muotoilut ja objektin elinkaaren. .NET-asiakasohjelman kannalta kaikki näyttää tavallisen hallitun objektin luonnilta ja sen *public*-tyyppisen jäsenen kutsumiselta!

Huomaa, että joka kerta, kun COM-metodi aiheuttaa poikkeuksen, RCW nappaa sen. Virhe muunnetaan sitten vastaavaksi *COMException*-luokaksi (löytyy nimiavaruudesta *System.Runtime.InteropServices*). COM-objektin pitää tietenkin yhä toteuttaa *ISupportErrorInfo* -rajapinta, jotta tämä virheen muunto onnistuisi ja jotta RCW tietää, että objektisi tuottaa virheinformaatiota. Virhe voidaan .NET-asiakasohjelmassa ottaa kiinni tavallisella *try-catch*-poikkeuksenkäsittelymekanismilla ja asiakasohjelma voi käsitellä virhenumeroa, kuvausta, poikkeuksen lähdettä ja muita yksityiskohtia, jotka olisivat minkä tahansa COM-kelpoisen asiakasohjelman käytettävissä. Jatketaan nyt tätä esimerkkiä hieman pidemmälle ja tutkitaan muita tapoja COM-komponenttien sitomiseen.

COM-rajapinnan valitseminen dynaamisesti

Miten perinteinen *QueryInterface*-menetelmä toimii .NET-asiakasohjelman kannalta, kun se haluaa käsitellä jotain toista COM-komponentin toteuttamaa rajapintaa? Teet siirtymisen toiseen rajapintaan yksinkertaisesti muuntamalla nykyisen objektisi tuohon tarvitsemaasi rajapintaan ja siinä kaikki! Sen jälkeen voit kutsua kaikkia haluamasi rajapinnan metodeja ja ominaisuuksia. Se on niin yksinkertaista.

RCW tekee jälleen suurimman osan työstä piilossa. Se on tavallaan sama asia, jolla Visual Basicin ajonaikainen ympäristö suojaa COM-asiakasohjelmien tekijöitä suoralta *QueryInterface*-rajapintaan liittyvän koodin kirjoittamiselta. Se yksinkertaisesti tekee *QueryInterface*-rajapinnan puolestasi, kun asetat objektin tyypin joksikin toisen objektin tyypiksi.

Katsotaan käytännössä, miten helppoa se on. Esimerkissä oletamme, että haluat kutsua *IAirportFacilities*-rajapinnan metodia, joka on toinen COM-objektimme toteuttama rajapinta. Voit tehdä sen muuntamalla (cast) *AirlineInfo*-objektin *IAirportFacilities*-rajapinnaksi. Sen jälkeen voit kutsua kaikkia metodeja, jotka kuuluvat *IAirportFacilities*-rajapintaan. Mutta ennen muunnosta haluat varmasti tarkistaa, tukeeko nykyinen objektisi tai toteuttaako se rajapinnan, jota aiot käyttää. Voit tehdä tarkistuksen *System.Type*-luokan *IsInstanceOf*-metodilla. Jos se palauttaa true, tiedät, että objekti tukee rajapintaa ja voit tehdä tyyppimuunnksen turvallisesti. Jos muunnat objektin joksikin mielivaltaiseksi rajapinnaksi, jota objekti ei tue, aiheutetaan *System.InvalidCastException*-poikkeus. Tällä tavalla RCW varmistaa, että teet tyyppimuunnoksen vain rajapintoihin, joita COM-objekti on toteuttanut. Koko koodi näyttää tältä:

```
using System;
using System.Runtime.InteropServices;
using System.Reflection;
using AIRLINEINFORMATIONLib;

public class AirlineClient2App
{
    public static void Main()
    {
        ////////////////////////////////////////////
        /// QUERY INTERFACE/ RT type Checking
        ////////////////////////////////////////////
        try
        {
           AirlineInfo objAirlineInfo;
           IAirportFacilitiesInfo objFacilitiesInfo;
```

```
            // Luodaan uusi AirlineInfo-objekti.
            objAirlineInfo = new AirlineInfo();

            // Kutsutaan GetAirlineTiming-metodia.
            String strDetails = objAirlineInfo.GetAirlineTiming
                                        (strAirline);

            // Kysellään IAirportFacilitiesInfo-rajapinta.
            objFacilitiesInfo =
                   (IAirportFacilitiesInfo)objAirlineInfo;

            //Kutsutaan IAirportFacilitiesInfo-rajapinnan
            //metodia.
            Console.WriteLine("{0}",
            objFacilitiesInfo.GetInternetCafeLocations());
        }
        catch(InvalidCastException eCast)
        {
            Console.WriteLine("We got an InvalidCast Exception " +
                              "- Message is {0}",eCast.Message);
        }
    }
}
```

Myöhäinen sidonta COM-komponentteihin

Kaksi tähän mennessä näkemääsi esimerkkiä (*AirlineClient1App* ja *AirlineClient2App*) käyttävät molemmat RCW-metadataa .NET-asiakasohjelman ja COM-objektin aikaiseen sidontaan. Aikaisella sidonnalla on koko joukko etuja, kuten vahva tyyppitarkistus käännöksessä, automaattiset tyyppitietojen esittäminen kehitystyökaluissa (kuten Visual Studio.NETissä) ja tietenkin parempi suorituskyky. Saattaa kuitenkin olla tilanteita, jolloin sinulla ei ole käännöksen aikana käytettävissä sen COM-komponentin metadataa, johon sitomista tarvitaan. Tällöin tarvitset *myöhäistä sidontaa* komponenttiin. Esimerkiksi jos komponentti, jota yrität käyttää, sisältää vain *dispinterface*-rajapinnan, sinulla ei juuri ole muuta mahdollisuutta kuin myöhäinen sidonta komponenttiin.

Voit toteuttaa myöhäisen sidonnan COM-komponenttiin reflection-menetelmän avulla, jota opit käyttämään luvussa 16. Jotta voit sitoa tällä tavalla COM-komponenttiin, sinun tulee tietää komponentin *ProgID*. Tämä johtuu siitä, että *System.Activator*-luokan staattinen metodi *CreateInstance* tarvitsee *Type*-objektin. Käyttämällä komponentin *ProgID*:tä voit kutsua *System.Type*-luokan *GetTypeFromProgID*-metodia. Tämä palauttaa kelvollisen .NETin *Type*-objektin, jota voit käyttää *System.Activator.CreateInstance*-metodin kutsussa. Kun olet tehnyt sen, voit käyttää mitä tahansa metodia tai ominaisuutta, jota komponentin

oletusrajapinta tukee käyttämällä sen *Type*-objektin *System.Type.InvokeMember*-instanssimetodia, jonka sait takaisin *GetTypeFromProgID*-metodilta.

Sinun pitää tietää vain metodin tai parametrin nimi ja sen hyväksymät parametrit. Kun kutsut myöhään sidotun komponentin metodia, välität parametrit sille kokoamalla ne yleiseen *System.Object*-taulukoon ja välittämällä sen metodille. Sinun tulee myös asettaa vastaavat *binding flags*-arvot riippuen siitä, oletko kutsumassa metodia tai asettamassa/lukemassa ominaisuuden arvoa.

Kuten näet seuraavassa koodissa, tehtävää on hieman enemmän kuin aikaisessa sidonnassa. Mutta tilanteissa, joissa myöhäinen sidonta on ainoa vaihtoehto, olet tyytyväinen, että se on mahdollista

```
using System;
using System.Runtime.InteropServices;
using System.Reflection;
using AIRLINEINFORMATIONLib;

public class AirlineClient3App
{
    public static void Main()
    {
        ////////////////////////////////////////////
        /// MYÖHÄINEN SIDONTA
        ////////////////////////////////////////////
        try
        {
           object objAirlineLateBound;
           Type objTypeAirline;

           object[] arrayInputParams= { “Air Scooby IC 5678” };

           objTypeAirline = Type.GetTypeFromProgID
                   (“AirlineInformation.AirlineInfo”);

           objAirlineLateBound = Activator.CreateInstance
                                  (objTypeAirline);

           String str = (String)objTypeAirline.InvokeMember
                                 (“GetAirlineTiming",
                                 BindingFlags.Default |
                                 BindingFlags.InvokeMethod,
                                 null, objAirlineLateBound,
                                 arrayInputParams);

            Console.WriteLine(“{0}",str);
```

```
            String strTime = (String)objTypeAirline.InvokeMember
                                        (“LocalTimeAtOrlando",
                                        BindingFlags.Default |
                                        BindingFlags.GetProperty,
                                        null, objAirlineLateBound,
                                        new object [] {});

            Console.WriteLine (“Hi there !. The Local Time in “ +
                                “Orlando,Florida is: {0}", strTime);
        }
        catch(COMException e)
        {
            Console.WriteLine(“Oops- We encountered an error “ +
                              "for Airline {0}. The Error message “ +
                              "is : {1}. The Error code is {2}",
                              strFoodJunkieAirline,
                              e.Message,e.ErrorCode);
        }
    }
}
```

COMin säikeistysmallit

Kun useimmat ihmiset aloittavat COM-ohjelmoinnin, heillä on hyvin vähän tai ei ollenkaan tietoa COMin säikeistysmalleista ja osastoista. Vasta kun he saavat lisää kokemusta, he oivaltavat, että vapaasti säikeistetty malli, jota he ovat käyttäneet, tuottaa suuria suorituskykyongelmia kun *single-threaded apartment* (*STA)* -asiakassäiettä käytetään luomaan *multithreaded apartment* (*MTA)*-objekti. Lisäksi uudet COM-ohjelmoijat harvemmin osaavat varautua säieturvallisuuteen ja vaaraan, joka odottaa heitä, kun samanaikaiset säikeet käsittelevät heidän COM-komponenttejaan.

Ennen kuin säie voi kutsua COM-objektia, sen tulee määritellä yhteytensä osastoon kertomalla aikooko se käyttää STA:ta vai MTA:ta. STA-asiakassäikeet kutsuvat joko *CoInitialize(NULL)* tai *CoInitializeEx(0, COINIT_APARTMENTTHREADED)* käyttääkseen STA:ta ja MTA-säikeet kutsuvat *CoInitializeEx(0, COINIT_MULTITHREADED)* käyttääkseen MTA:ta. Vastaavasti .NETin hallitsemassa maailmassa sinulla on mahdollisuus antaa kutsuvan säikeen määritellä osastoyhteytensä. Oletuksena hallitun sovelluksen kutsuva säie valitsee osastokseen MTA:n. Se on sama kuin kutsuva säie alustaisi itsensä metodilla *CoInitializeEx(0, COINIT_MULTITHREADED)*. Mutta ajattele ylimääräistä työtä ja heikentynyttä suorituskykyä, johon joudutaan, jos se kutsui tavallista STA COM-komponenttia, joka on suunniteltu olemaan *apartment-threaded*. Yhteensopimattomat osastot aiheuttavat ylimääräisen proxy/stub-parin generoinnin ja se aiheuttaa ilman muuta suorituskyvyn laskua.

Siksi voit .NET-sovelluksessa korvata hallitun säikeen osaston oletusvalinnan käyttämällä *System.Threading.Thread*-luokan *ApartmentState*-ominaisuutta. Se asetetaan joksikin seuraavista luetelluista tyypeistä:

- ***MTA*** Monisäikeinen osasto
- ***STA*** Yksisäikeinen osasto
- ***Unknown*** Vastaava kuin oletuksena oleva MTA

Sinun pitää myös määritellä kutsuvan säikeen *ApartmentState*-ominaisuus ennen kuin teet kutsuja COM-objektiin. Huomioi, että *ApartmentState*-ominaisuutta ei ole mahdollista muuttaa sen jälkeen, kun COM-objekti on luotu. Siksi on järkevää asettaa säikeen *ApartmentState*-ominaisuus koodissa niin aikaisin kuin mahdollista. Seuraava koodi näyttää, miten se tehdään:

```
// Asetetaan säikeen ApartmentState arvoon STA.
Thread.CurrentThread.ApartmentState =
    ApartmentState.STA;

// Luodaan COM-objekti Interopin kautta.
MySTA objSTA = new MySTA();
objSTA.MyMethod()
```

Yhteenveto

Kerroin tässä luvussa miten vanhan koodin käsittelyn eri mekanismit (PInvoke, turvaton koodi ja COM Interop) sopivat ajatelleen .NETiä kokonaisuutena. Tässä luvussa opit seuraavaa:

- Tavallisiin C-tyyppisiin funktiokutsuihin liittyen opit, miten käytät *PInvoke*-metodia yhdessä eri attribuuttien kanssa, jotka tekevät eri tyyppisten tietojen muotoilun helpommaksi.sta.
- Turvattomaan koodiin liittyen opit, miten luopua hallitun koodin eduista C#-ohjelmassa tilanteissa, joissa tarvitset täsmällisempää hallintaa. Näitä tilanteita ovat mm. se, kun haluat itse hallita muistia tehokkuuden maksimoimiseksi tai kun siirrät koodilohkoja C#-sovellukseen etkä ole vielä valmis muuntamaan sitä hallituksi koodiksi.

- COMiin liittyen näit, miten käytät perinteistä COM-komponenttia .NET-sovelluksessa ja miten COM Interop mahdollistaa saumattomasti olemassa olevien COM-komponenttien käytön hallitusta koodista. Sitten kävimme läpi, miten käytetään COM-komponentteja sekä aikaisella sidonnalla että myöhäisellä sidonnalla yhdessä suorituksen aikaisen tyyppitarkistuksen kanssa. Lopuksi näit miten hallitut säikeet määrittelevät osastoyhteytensä, kun ne käyttävät COM-komponentteja.

Tässä vaiheessa ohjelmoija, joka soveltaa jotakin näistä menetelmistä käyttäessään hallitsematonta koodia, saattaa miettiä, onko järkeä jatkaa näiden menetelmien kanssa vai olisiko parempi siirtyä suoraan .NET-maailmaan kirjoittamalla kaikki komponentit ja liiketoimintalogiikkakoodit käyttäen C#:n tapaista .NET-kieltä. Ratkaisu riippuu täysin tilanteesta.

Jos sinulla on paljon vanhaa koodia (on se sitten C-kielen tapaisia funktioita DLL:ssä, koodia, joka käsittelee suoraan muistia, COM-komponentteja tai näiden kolmen yhdistelmiä), tosiasia on, että et pysty muuntamaan kaikkea koodiasi hetkessä. Tässä tapauksessa on järkeä hyödyntää erilaisia .NET-mekanismeja ja käyttää vanhaa koodia. Jos kuitenkin kirjoitat uutta koodia alusta alkaen, suosittelen lämpimästi, että teet sen käyttäen hallittuja komponentteja ja koodaat ne C#:n tapaisella .NET-kielellä. Tällä tavalla voit välttää suorituskyvyn heikkenemisen, joka väistämättä tapahtuu, kun siirrytään hallitsemattoman ja hallitun koodin raja-aidan yli.