3

Hello, C#

Ennen kuin menemme asiamme varsinaiseen ytimeen (osaan II, ”C#-luokkien perusteet” ja osaan III, ”Koodin kirjoittaminen”) ajattelin kirjoittaa eräänlaisen ”Näin pääset alkuun” -luvun. Tässä luvussa käymme nopeasti läpi yksinkertaisen C#-sovelluksen teon vaiheet. Ensinnäkin kerron niiden editorien eduista ja haitoista, joilla voit kirjoittaa C#-sovelluksia. Kun olet valinnut oman editorisi, teemme pakollisen ”Hello, World” -esimerkkisovelluksen, jotta saat käsityksen C#-sovelluksen teon perussyntaksista ja rakenteesta. Huomaat, että kuten muissakin kielissä, syntaksi on kaavamainen ja voit käyttää esimerkkisovellusta useimpien tavallisten C#-sovellustesi pohjana. Sen jälkeen opit kääntämään ohjelmasi komentorivikääntäjällä ja lopuksi opit käynnistämään uuden sovelluksesi.

Ensimmäisen C#-sovelluksen kirjoittaminen

Käydään läpi ne vaiheet, joilla saat ensimmäisen C#-sovelluksesi tehtyä ja käynnistettyä.

Editorin valinta

Ennen kuin kirjoitat C#-sovelluksen, sinun pitää valita editori. Seuraavat kappaleet esittelevät yleisimmät editorit ja kertovat niistä eräitä tosiasioita, jotka vaikuttavat niiden valintaan C#-editoriksi.

Muistio

Microsoftin Muistio on ollut C#:n alkuaikoina yleisin editori niiden ohjelmoijien joukossa, jotka ovat käyttäneet .NET Framework SDK:ta C#-sovellusten tekemiseen. Itse käytin Muistiota tätä kirjaa tehdessäni eräistä syistä, jotka kerron myöhemmin. En kuitenkaan suosittele Muistion käyttämistä seuraavalla sivulla lueteltujen syiden vuoksi.

- C#-tiedostot tulee tallentaa .cs-tarkenteella. Jos et ole huolellinen, Muistion Tallenna Nimellä -valintaikkunassa käy helposti niin, että kun yrität tallentaa nimellä Test.cs, tiedoston nimeksi tuleekin test.cs.txt. Sinun pitää muistaa ennen tallennusta muuttaa Tallennusmuoto-luetteloon kohta "Kaikki tiedostot (*.*)".
- Muistio ei näytä rivinumeroita ja se on iso juttu, kun kääntäjä kertoo virheestä määrätyllä rivillä.
- Muistion sarkaimen leveys on kahdeksan välilyöntiä, jolloin kaiken muun kuin pelkän "Hello, Word" -ohjelman tekeminen aiheuttaa vaikeasti luettavaa koodia.
- Muistio ei suorita automaattista sisennystä, kun painat Enter-näppäintä. Siksi sinun pitää käsin siirtyä haluamaasi sarakkeeseen koodin kirjoittamista varten.

Muitakin syitä on olemassa, mutta kuten näet, Muistio ei ole hyvä valinta C#-sovellusten tekemiseen.

Visual Studio 6

Koska Microsoft Windows -ohjelmointitaustani on Microsoft Visual C++-kielessä, on Microsoft Visual Studio 6 luonnollinen valintani editoriksi. Visual Studio on monipuolinen editori, joka sisältää kaikki tarpeelliset ominaisuudet C#-tiedostojen muokkaamiseen ja tallentamiseen.

Yksi suurimmista eduista ohjelmointiin tarkoitetun editorin käytössä on sen kyky rakenteen erotteluun esimerkiksi värein. Koska Visual Studio julkaistiin muutama vuosi ennen C#:ia ja se suunniteltiin Visual C++ sovellusten tekoon, sinun pitää hieman muokata sitä, jotta se tunnistaisi C#-syntaksin. Ensimmäinen askel on muuttaa Visual Studion rekisteriavainta. Etsi seuraava avain rekisteristäsi käyttämällä Regedit.exe-ohjelmaa tai jotain muuta rekisterin muokkaamiseen tehtyä ohjelmaa:

HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\Text Editor\Tabs/Language Settings\C/C++\FileExtensions

Arvo sisältää seuraavanlaisen merkkijonon:

cpp;cxx;c;h;hxx;hpp;inl;tlh;tli;rc;rc2

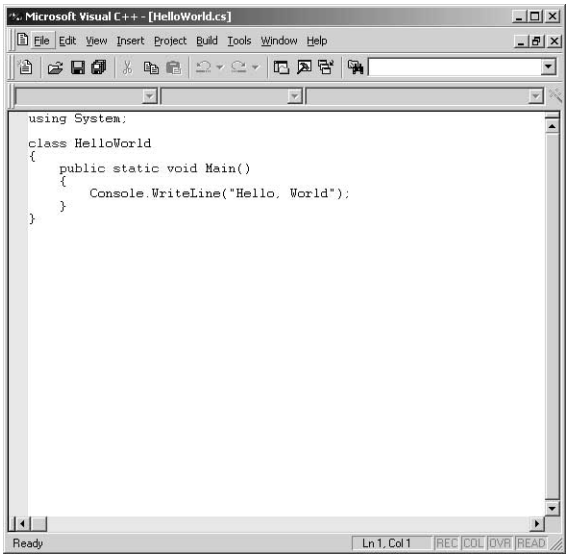
Lisää arvon loppuun *.cs*-tunniste. (Huomaa, että lopun puolipiste ei ole pakollinen.) Rekisterin uusi arvo näyttää seuraavalta:

cpp;cxx;c;h;hxx;hpp;inl;tlh;tli;rc;rc2;cs

Nyt kun avaat sellaisen tiedoston Visual Studio 6:ssa, jonka tunniste on .cs, Visual Studio tunnistaa sen tuetuksi tiedostoksi.

Seuraavaksi sinun tulee kertoa Visual Studiolle mitkä ovat C#:n avainsanat. Teet sen luomalla tiedoston nimeltä usertype.dat ja sijoittamalla sen samaan kansioon, jossa

msdev.exe-tiedosto on. Se on ASCII-tiedosto, joka sisältää kaikki ne avainsanat, jotka tulee värittää. Kirjoita yksi avainsana aina yhdelle riville. Kun Visual Studio käynnistyy, se lukee tämän tiedoston. Kun teet tiedostoon muutoksia, sinun tulee käynnistää Visual Studio uudelleen, jotta se ottaisi muutokset huomioon. Tämän kirjan mukana tulevalla cd-levyllä on sellainen versio usertype.dat-tiedostosta, joka sisältää C#-kielen avainsanat. Kuva 3-1 näyttää, miltä C#-koodisi näyttää, kun olet seurannut edellä kerrottuja ohjeita.



Kuva 3-1 Syntaksin merkinnän tarjoavan Visual Studio 6:n editorin yksi etu on se, että saat välittömän palautteen avainsanan kelpoisuudesta.

Visual Studio.NET

On aivan selvää, että jos haluat kaiken tehokkuuden irti .NET-ympäristöstä, sinun tulee käyttää Visual Studio.NETiä. Se ei tarjoa pelkästään kaikkia mielenkiintoisia integroituja työkaluja ja wizardeja C#-sovellusten tekoon, vaan myös tuottavuutta lisääviä ominaisuuksia kuten IntelliSense ja Dynamic Help. IntelliSense näyttää automaattisesti luokan tai nimiavaruuden jäsenet, kun kirjoitat niiden nimen editoriin, eikä sinun tarvitse yrittää muistella jäseniä. IntelliSense näyttää myös kaikki parametrit ja niiden tyypin, kun kirjoitat metodin nimen ja avaavan sulun. Visual Studio 6:ssa on myös tämä ominaisuus, mutta se ei tietenkään tue .NETin tyyppejä ja luokkia. Dynamic Help on Visual Studion uusi ominaisuus. Kun olet kirjoittamassa koodia editoriin, näyttää erillinen ikkuna sanaan

liittyviä avusteen aiheita. Jos esimerkiksi kirjoitat avainsanan *namespace*, ikkuna näyttää hyperlinkin niihin avusteen aiheisiin, jotka kuvaavat *namespace*-avainsanan käyttöä.

Muiden tekemät editorit

Eipä unohdeta, että on myös olemassa paljon suosittuja jonkun muun kuin Microsoftin tekemiä editoreja, kuten Starbase-yhtiön CodeWright ja MicroEdge-yhtiön Visual SlickEdit. En mene niiden yksityiskohtiin, mutta totean, että voit aivan hyvin käyttää C#-sovellusten tekoon mitä editoria haluat.

Mitä itse käytin tätä kirjaa tehdessäni

Koska aloitin tämän kirjan tekemisen silloin, kun Visual Studio.NET oli vielä testausvaiheessa, käytin Visual Studio 6:tta. Toivon, että sellaisen ympäristön käyttäminen, joka ei ole suunniteltu C#-ohjelmien tekoon, auttoi minua pitämään lupaukseni, että tämä kirja on käyttökelpoinen jokaiselle C#-sovellusten tekijälle riippumatta hänen valitsemastaan kehitysympäristöstä. Kuten aiemmin mainitsin, voit kirjoittaa tämän kirjan esimerkit jopa Muistiolla ja saat niistä aivan samat tulokset.

Hello, World

Oletetaan, että olet nyt valinnut kehitysympäristön ja siirrytään ensimmäiseen C#-sovellukseemme, siihen pakolliseen ”Hello, World”-sovellukseen. Kirjoita tiedostoon seuraava koodi ja tallenna se nimelle HelloWorld.cs:

```
class HelloWorld
{
    public static void Main()
    {
        System.Console.WriteLine(“Hello, World”);
    }
}
```

Älä huolehdi vielä tässä vaiheessa siitä, mitä kukin rivi tekee. Tässä vaiheessa keskitymme kirjoittamaan ensimmäisen sovelluksemme sekä saamaan sen käännettyä ja suoritettua. Kun olemme tehneet sen (joka todistaa, että sinulle on kelvollinen ympäristö C#-sovellusten tekemiseen ja suorittamiseen) katsomme koodin yksityiskohtia tarkemmin.

Komentorivikääntäjän käyttäminen

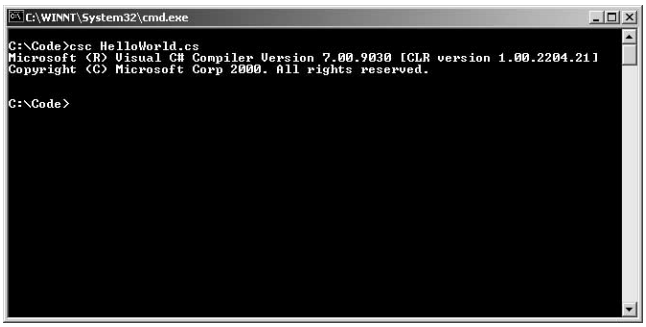
Jos käytät editoria, jossa on sisäänrakennettu C#-sovelluksen kääntämistoiminto, tämä vaihe saattaa tuntua kömpelöltä. Mutta jotta pysyisimme niin editori-riippumattomina kuin mahdollista, tulen käyttämään C#:n komentorivikääntäjää (csc.exe) koko kirjan ajan. Tästä on kaksi etua. Ensinnäkään ei ole mitään merkitystä, mitä ympäristöä käytät,

esimerkkiohjelman teon vaiheet pätevät. Toiseksi, erilaisten käännösvalitsimien oppiminen auttaa sinua pitkällä tähtäimellä tilanteissa, joissa kehitysympäristösi ei salli tämän vaiheen täydellistä hallitsemista.

Kun olet kirjoittanut ”Hello, World”-ohjelman koodin, avaa DOS-ikkuna ja siirry kansioon, jossa HelloWorld.cs-tiedosto sijaitsee. Kirjoita seuraava:

```
csc helloWorld.cs
```

Jos kaikki toimi oikein, sinun pitäisi nähdä vastaavat tulokset kuin kuvan 3-2 ikkunassa. Siinä näet kääntäjän nimen ja version sekä mahdolliset virheilmoitukset tai varoitukset. Kuten näet, käytin kääntäjän beeta-versioita silloin, kun kirjoitin tätä. Huolimatta käyttämäsi kääntäjän versiosta, sinun ei pitäisi nähdä yhtään virhettä tai varoitusta tämän yksinkertaisen esimerkin käännöksen tuloksena.



Kuva 3-2 HelloWorld.cs:n kääntämisen ei pitäisi aiheuttaa virheitä tai varoituksia.

Jos saat seuraavanlaisen virheilmoituksen *'csc' is not recognized as an internal or external command, operable program or batch file*, se luultavasti tarkoittaa, että et ole asentanut .NET SDK:ta, joka sisältää C#-kääntäjän.

Jos käynnistät C#-kääntäjän väärin (annat esimerkiksi väärän tiedostonimen) tai kutsut sitä /?-valitsimella, kääntäjä näyttää kaikki mahdolliset valitsimet, jotka voit asettaa sovelluksesi käännökselle. En halua juuttua selittämään mitä kukin yksittäinen asetus tarkoittaa, sen sijaan keskityn asetuksiin, jotka ovat tärkeitä kokonaisuuden kannalta ja voit tutustua loppuihin vapaa-aikanasi.

Sovelluksen käynnistäminen

Nyt kun olemme kääntäneet ”Hello, World”-sovelluksen, käynnistetään se, jotta saamme varmuuden, että .NETin ajonaikainen ympäristö on kunnossa. Kaikki tämän kirjan

esimerkit ovat ”konsolisovelluksia” eli että me keskitymme C#-kieleen emmekä mihinkään erityiseen Windows-sovelluskehitykseen. Tämän vuoksi voit käynnistää esimerkit käskyriviltä tai editoristasi, jos se tukee komentorivisovellusten suorittamista.

Jos käytät komentorivivaihtoehtoa, avaa nyt DOS-ikkuna, siirry siihen kansioon, jonne käänsit esimerkin ja kirjoita **HelloWorld**. Tuloksen pitäisi näyttää seuraavalta:

```
d:\>HelloWorld
Hello, World
```

Jos toisaalta yrität käynnistää sovelluksen editorista, etkä näe mitään tai DOS-ikkuna avautuu, suorittaa sovelluksen ja katoaa nopeasti, niin muuta sovellusta seuraavasti:

```
class HelloWorld
{
    public static void Main()
    {
        System.Console.WriteLine(“Hello, World”);
        String str = System.Console.ReadLine();
    }
}
```

Funktiokutsu *System.Console.ReadLine* aiheuttaa sen, että sovellus pysähtyy kunnes painat ENTER-näppäintä ja siksi pystyt varmistamaan, että sovellus on kirjoittanut sen minkä pitikin. Tästä eteenpäin esimerkkiohjelmat eivät sisällä tätä koodiriviä. Jos käynnistät sovelluksesi suoraan editorista, sinun pitää muistaa lisätä edellä oleva rivi jokaiseen sovellukseen ennen loppua.

Koodin tutkiminen

Nyt kun olemme todistaneet, että pystyt kirjoittamaan ja suorittamaan C#-sovelluksen, käykäämme koodi läpi tarkemmin ja tutkikaamme C#-sovelluksen perusrakenne.

Yhden pysäyksen ohjelmointi

Kuten olet nähnyt edellisissä luvuissa ja ”Hello, World”-sovelluksessa, esimerkkini ovat osoittaneet, että kunkin luokan metodit määritellään itse luokan määrittelyn sisällä. Tämä ei ole minun puoleltani ainoastaan mukavuudenhalua, kuten C++ ohjelmoijat voisivat ajatella. Kun teet ohjelmia C++:lla, sinulla on kaksi vaihtoehtoa: voit ohjelmoida luokan jäsenfunktion toteutuksen suoraan luokan määrittelyn sisään (esimerkki *inline-funktiosta*) tai voit erottaa luokan määrittelyn ja sen jäsenfunktion toteutuksen erillisiin tiedostoihin. C#:ssa sinulla ei ole tätä valinnanmahdollisuutta.

Kun C#:ssa määrittelet luokan, sinun tulee määritellä kaikki sen metodit luokan sisällä (inline), otsikkotiedostoja ei ole olemassa. Miksi tämä on hyvä asia? Koska se mahdollistaa, että luokkien tekijät luovat helposti liikuteltavaa koodia, joka on yksi .NET-ympäristön avainominaisuus. Tällä tavoin, kun kirjoitat C#-luokan, teet täysin kapseloidun ”nipun” toiminnallisuutta, jonka voit helposti siirtää mihin tahansa kehitysympäristöön välittämättä siitä, miten sen kieli käsittelee include-tiedostoja tai onko siinä menetelmä tiedostojen lukemiseen toisen sisään. Tämän ”yhden pysäyksen ohjelmoinnin” -periaatteen käyttämisellä voit esimerkiksi ottaa kokonaisen luokan ja pudottaa sen Active Server Pages (ASP) -sivulle ja se toimii aivan samoin kuin se toimii, jos se olisi käännetty Windows-sovellukseksi!

Luokat ja jäsenet

C#:lla tehdyssä perussovelluksessa näet ensin luokan tai nimiavaruuden nimen. Kuten opit luvussa 1, ”Olio-ohjelmoinnin perusteet”, sinun tulee valita luokalle nimi, joka kuvaa käsitettä, esimerkiksi *Lasku*, *Tilaus* tai *Asiakas*. Luokan määrittelyn alku ja loppu merkitään aaltosuluilla { ja }. Kaikki niiden välillä oleva käsitetään osaksi tuota C#-luokkaa. Huomaa esimerkissä, että meillä on luokka nimeltä *HelloWorld* ja kaikki sovelluksessa oleva määritellään sen sisällä.

Kaikki luokan jäsenet määritellään luokan aaltosulkeiden välissä. Näitä jäseniä ovat metodit, kentät, ominaisuudet, indeksoijat, attribuutit ja rajapinnat. Kerron seuraavissa luvuissa, miten nämä C#-luokan eri elementit määritellään.

Main-metodi

Jokaisessa C#-ohjelmassa pitää olla *Main*-niminen metodi määriteltynä jossain sen luokista. Sillä ei ole merkitystä, missä luokassa metodi on (sinulla voi yhdessä sovelluksessa olla niin monta luokkaa kuin tarvitset), kunhan yhdessä luokassa on *Main*-niminen metodi. Lisäksi tämä metodi tulee määritellä sekä *julkiseksi* (public) että *staattiseksi* (static). *Public* on käsittelymääre, joka kertoo C#-kääntäjälle, että kuka tahansa voi kutsua tätä metodia. Kuten näit luvussa 1, avainsana *static* kertoo kääntäjälle, että *Main*-metodi on *yleinen* (global) metodi ja että luokkaa ei tarvitse instantioida, jotta metodia voisi kutsua. Tässä on järkeä, kun ajattelet sitä, koska muuten kääntäjä ei tietäisi miten tai koska sen tulisi luokka instantioida. Koska metodi on staattinen, kääntäjä tallentaa metodin osoitteen alkukohtana (entry point), jotta .NETin ajonaikainen ympäristö tietää, mistä aloittaa sovelluksesi suorittamisen.

> Huomaa Tämän luvun esimerkit osoittavat, että *Main*-metodin paluuarvo on *void* ja että se ei saa parametreja. Voit kuitenkin määritellä, että *Main*-metodisi palauttaa arvon sekä ottaa vastaan taulukon parametreja. Nämä vaihtoehdot, sekä se, miten käyt läpi sovelluksen *Main*-metodille välitettyä parametritaulukkoa, käsitellään luvussa 4, ”Luokat”.

System.Console.WriteLine-metodi

System.Console.WriteLine-metodi kirjoittaa annetun merkkijonon ja rivinvaihtomerkin oletustulostuslaitteelle. Useimmissa tapauksissa, jos et tee jotain hauskoja asetuksia tai käytät editoria, joka ohjaa tulosteen ikkunalle, tämä tarkoittaa, että merkkijono tulostetaan DOS-ikkunaan.

Nimiavaruudet ja *using*-määre

Luvussa 2, ”Johdanto Microsoft NETiin”, opit, että .NET Framework luokkakirjasto on järjestetty hierarkkisiin nimiavaruuksiin. Tämä saattaa johtaa melko pitkiin nimiin, kun tarvittava luokka tai tyyppi sijaitsee hierarkiassa monen kerroksen takana. Kirjoittamisvaivan vähentämiseksi C# sisältää *using*-määreen. Katsotaanpa esimerkkiä määreen toiminnasta. ”Hello, World”-sovelluksessamme meillä on seuraava koodirivi:

```
System.Console.WriteLine(“Hello, World”);
```

Tämän kirjoittaminen kerran ei ole iso juttu, mutta kuvittele, että sinun pitäisi kirjoittaa jokaisen tyypin ja luokan täysin määritelty nimi isossa sovelluksessa. *using*-määre antaa sinulle mahdollisuuden tehdä eräänlainen hakupolku niin, että jos kääntäjä ei ymmärrä jotakin, mitä olet kirjoittanut, se etsii määrittelemästäsi hakupolusta. Kun sovellamme *using*-määrettä sovellukseemme, saamme sen seuraavaan muotoon:

```
using System;

class HelloWorld
{
    public static void Main()
    {
        Console.WriteLine(“Hello, World”);
    }
}
```

Kun kääntäjä kohtaa *Console.WriteLine*-metodin, se toteaa, että metodia ei ole määritelty. Se kuitenkin etsii *using*-määreellä määritellystä nimiavaruudesta ja kun on löytänyt metodin *System*-nimiavaruudesta, kääntää koodin ilman virhettä.
Huomaa, että *using*-määrettä sovelletaan nimiavaruuksiin, ei luokkiin. Esimerkissämme *System* on nimiavaruus, *Console* on luokka ja *WriteLine* on staattinen metodi, joka kuuluu *Console*-luokkaan. Siksi seuraava koodi olisi virheellinen:

```
using System.Console; //VIRHE Et voi käyttää
                      //using-määrettä luokan kanssa.

class HelloWorld
{
    public static void Main()
    {
        WriteLine(“Hello, World”);
    }
}
```

Vaikka et voi määritellä luokkaa *using*-määreessä, voit seuraavan *using*-määreen muunnoksen avulla antaa luokalle peitenimen (aliaksen):

using *alias* = *class*

Käyttäen tätä *using*-määreen muotoa, voit kirjoittaa koodin seuraavasti:

```
using ouput = System.Console;

class HelloWorld
{
    public static void Main()
    {
        output.WriteLine(“Hello, World”);
    }
}
```

Tämä antaa sinulle joustavuutta soveltaa järkeviä peitenimeä luokkiin, jotka sijaitsevan muutamia tasoja alempana .NETin nimiavaruus-hierarkiassa. Näin saat koodistasi hieman helpomman kirjoittaa ja ylläpitää.

Koodin runko

Katsotaan nopeasti mitä tämän esimerkin koodissa on sellaista, jota voi käyttää useimmissa C#-sovelluksissa eli koodia, joka muodostaa jokaisen tavallisen C#-sovelluksen rungon. Se kannattaa kirjoittaa tiedostoon ja tallentaa, jotta voit käyttää sitä myöhemmin pohjana. Seuraavassa runkokoodissa kulmasulut ilmoittavat paikan, minne sinun pitää kirjoittaa jotain.

```
using <namespace>
namespace <your optional namespace>
class <your class>
{
    public static void Main()
    {
    }
}
```

Luokan moniselitteisyys

Jos tyyppi on määritelty useammassa kuin yhdessä viitatussa nimiavaruudessa, kääntäjä ilmoittaa moniselitteisyysvirheen. Siksi seuraava koodi ei käänny, koska luokka *C* on määritelty kahdessa nimiavaruudessa, joihin molempiin viitataan *using*-määreessä:

```
using A;
using B;

namespace A
{
    class C
    {
        public static void foo()
        {
            System.Console.WriteLine("A.C.foo");
        }
    }
}

namespace B
{
    class C
    {
        public static void foo()
        {
            System.Console.WriteLine("B.C.foo");
        }
    }
}

class MultiplyDefinedClassesApp
{
    public static void Main()
    {
        C.foo();
    }
}
```

Tämän tyyppisen virheen välttämiseksi varmista, että luokkiesi ja metodiesi nimet ovat yksilöllisiä ja kuvaavia.

Jotain meni pieleen!

Mitä tiedämme tähän asti? Tiedämme, miten kirjoitetaan, käännetään ja ajetaan C#-sovelluksia ja meillä on peruskäsitys C#-sovelluksen rakenteesta. Entä jos hyvälle C#-sovellukselle tapahtuu pahoja asioita? Määritellään ensin paha asia: mitä tahansa sellaista, jota et odottanut tapahtuvaksi. Ohjelmoinnin ollessa kyseessä nämä voidaan jakaa kahteen luokkaan: kääntäjän huomaamiin virheisiin ja suorituksen aikaisiin virheisiin. Katsotaan muutama esimerkki kummastakin ja miten ne tulee korjata.

Kääntäjän ilmoittamat virheet

Kun kääntäjä, myös C#-kääntäjä, ei pysty tulkitsemaan, mitä yrität koodillasi tehdä, se tulostaa virheilmoituksen ja sovelluksesi käännös epäonnistuu. Kirjoita seuraava koodi HelloErrors.cs-nimiseen tiedostoon ja käännä se:

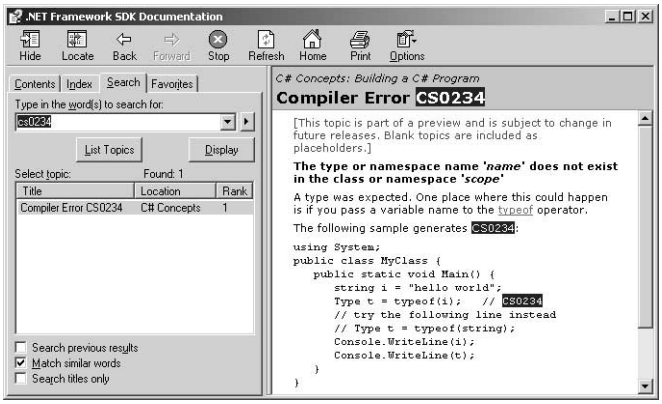
```
using Syste;

class HelloErrors
{
    public static void Main()
    {
        xConsole.WriteLine(“Hello, World”);
        Console.WriteLinex(“Hello, World”);
    }
}
```

Kääntäjä ilmoittaa seuraavaa:

```
HelloErrors.cs(1,7): error CS0234: The type or namespace name
‘ Syste’ does not exist in the class or namespace ‘‘
```

Pitäen mielessä, että aina on olemassa oletusnimiavaruus, virheilmoitus tarkoittaa, että kääntäjä ei ilmiselvästä syystä pysty löytämään mitään Syste-nimistä. Haluan tällä tuoda esille sen, mitä on odotettavissa, kun kääntäjä löytää koodistasi syntaksivirheen. Näet ensinnäkin käännettävän tiedoston nimen, jota seuraa virheen paikan rivi- ja sarakenumero. Seuraavana näet kääntäjän määrittelemän virhekoodin, tässä tapauksessa *CS0234*. Virhekoodin jälkeen näet lyhyen kuvauksen virheestä. Monta kertaa tämä kuvaus kertoo sinulle tarpeeksi virheen aiheuttajasta. Jos se ei kuitenkaan riitä, voit etsiä virhekoodin tarkemman selostuksen .NET Framework SDK Documentation -ohjelmasta, joka asennetaan .NET Framework SDK:n mukana. Virhekoodiin CS0234 liitetty on-line avusteen teksti on seuraavan sivun kuvassa 3-3.



Kuva 3-3 Voit käyttää C#-kääntäjän ilmoittamaa virhekoodia hakiessasi lisäselvitystä on-line-avusteelta.

Huomaa, että vaikka teimme kolme virhettä (*System*-nimiavaruus oli väärin kirjoitettu, *Console*-luokka oli väärin kirjoitettu ja *WriteLine*-metodin kutsu oli väärin kirjoitettu), kääntäjä ilmoitti vain yhden virheen. Tämä johtuu siitä, että kun määrätty virhe kohdataan, kääntäjä lopettaa käännöksenä siihen paikkaan ja tulostaa siihen mennessä kertyneet virheet. Tässä esimerkissä kääntäjä lopetti toimintansa, kun se ei pystynyt ratkaisemaan *using*-määrettä, koska sellainen virhe olisi voinut aiheuttaa monia muita virheitä. Kun olet korjannut *System*-nimiavaruuden kirjoituksen *using*-määreessä, kääntäjä ilmoittaa rivi- ja sarakenumeron myös kahden jäljellä olevan virheen osalta.

Tutkiminen ILDASM-ohjelmalla

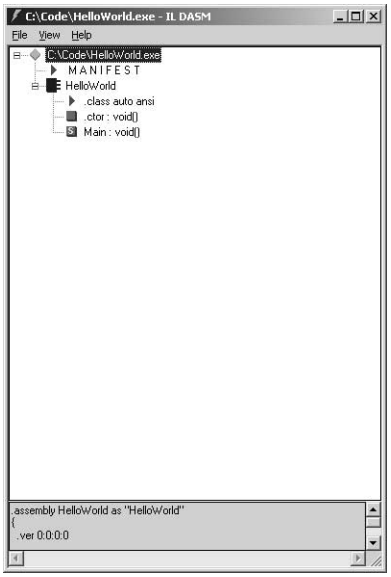
Kuten luit luvussa 2, kun luot EXE- tai DLL-tiedoston käyttämällä .NET-kääntäjää, ei tiedosto ole tavallinen ajettava ohjelma. Sen sijaan se koostuu luettelosta (manifest), joka sisältää luettelon tiedostoon sisältyvistä tyypeistä ja luokista ja MSIL (Microsoft intermediate language) -välikoodin, jonka myöhemmin kääntää ja suorittaa asennusohjelmasi tai .NETin ajonaikainen ympäristö täsmäkäännöksenä (just-in-time) JITerillä.

Suuri etu tästä on se, että generoitu MSIL näyttää aivan assemly-kieleltä ja sitä voidaan käyttää erinomaisena opetustyökaluna esittelemään, mitä kääntäjä on tehnyt koodillemme. Tämän vuoksi palaan monta kertaa tässä kirjassa C#-kääntäjän tekemään MSIL-koodiin esittäessäni, miten jokin toimii kulissien takana tai selittäessäni, miksi tulee käyttää määrättyä kielen ominaisuutta määrätyllä tavalla. Jotta .NET-kääntäjän tekemän MSIL:n voisi nähdä, Microsoft on sisällyttänyt SDK:hon Microsoft .NET Framework IL Disassembler

(ILDASM)-nimisen ”MSIL-tulkin”, jonka avulla voit avata .NETin suoritustiedostoja (EXE tai DLL) ja tutkia sen nimiavaruuksia, luokkia, tyyppejä ja koodia. Aloitamme ILDASMiin tutustumisen seuraavassa kappaleessa.

Hello, World MSIL-koodina

Kirjoita Käynnistä-valikon Suorita-valintaikkunan Avaa-kenttään **ildasm** ja napauta OK-painiketta. Näet dokumentoimattoman sovelluksen, jossa on muutamia valikkokohtia. Valitse File-valikon kohta Open. Siirry File Open -valintaikkunassa aiemmin tekemäsi (sivulla 39) HelloWorld.exe-sovelluksen kansioon ja valitse se. Kuten näet kuvassa 3-4, asia alkaa näyttää lupaavalta.



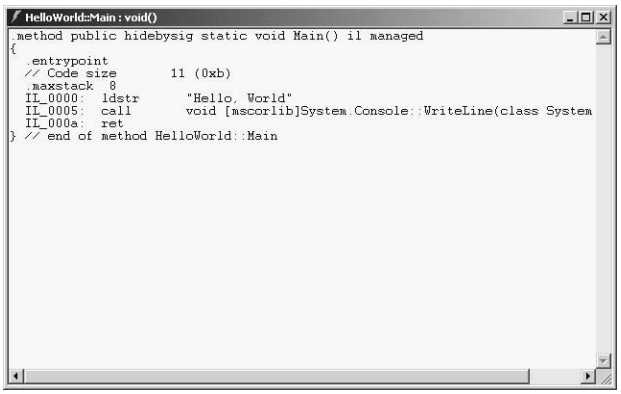
Kuva 3-4 ILDASMin avulla voit tutkia luetteloa ja IL:n välikoodia, jotka muodostavat .NET-sovelluksesi.

Huomaa puumainen rakenne, jota ILDASM käyttää kuvatessaan hallittua binaaritiedostoa. Kuvassa 3-5 ovat eri kuvakkeet, joita ILDASM puu-kontrollissa käyttää kuvaamaan .NET -sovelluksen osia. Kuten näet kuvia 3-5 ja 3-4 tutkimalla, HelloWorld.exe koostuu luettelosta, yhdestä luokasta (*HelloWorld*), kahdesta metodista (luokan muodostimesta ja staattisesta *Main*-metodista) ja luokan tiedoista.

	Nimiavaruus:	**Sininen alue**
	Luokka:	**Sininen suorakaide kolmella ulostulolla**
	Rajapinta:	**Sininen suorakaide kolmella ulostulolla, jotka on merkitty kirjaimella I**
	Arvoluokka:	**Ruskea suorakaide kolmella ulostulolla**
	Enum:	**Ruskea suorakaide kolmella ulostulolla, jotka on merkitty kirjaimelle E**
	Metodi:	**Purppura suorakaide**
	Staattinen metodi:	**Purppura suorakaide, joka on merkitty kirjaimella S**
	Kenttä:	**Sinertävä kuusikulmio**
	Staattinen kenttä:	**Sinertävä kuusikulmio, joka on merkitty kirjaimella S**
	Tapahtuma:	**Vihreä alas osoittava kolmio**
	Ominaisuus:	**Punainen ylös osoittava kolmio**
	Luettelo tai luokan informaatio:	**Punainen oikealle osoittava kolmio**

Kuva 3-5 Kuvakkeet, joilla ILDASMissa ilmaistaan .NET-sovelluksen eri osat.

Mielenkiintoisin osa ”Hello, World”-sovellusta on *Main*-metodi. Kaksoisnapauta sitä ILDASM-puussa, jolloin ILDASM näyttää ikkunassa *Main*-metodin MSIL:n, kuten kuvassa 3-6.



Kuva 3-6 Kun haluat tutkia metodin generoitua MSIL:ää, avaa ohjelman binaaritiedosto ILDASM:ssa ja kaksoisnapauta metodia.

”Hello, World” ei ole edes MSIL:nä kovin mielenkiintoinen, mutta voit oppia generoidusta MSIL:stä muutamia tosiasioita, jotka pätevät mihin tahansa .NET-sovellukseen. Tutkitaan metodia rivi riviltä, niin huomaat, mitä tarkoitan.

```
.method public hidebysig static void Main() il managed
{
    .entrypoint
    // Code size    11 (0xb)
    .maxstack 8
    IL_0000:         ldstr     “Hello, World”
    IL_0005:    call void [mscorlib]System.Console::WriteLine
                        (class System.String)
} // end of method HelloWorld::Main
```

Ensimmäinen rivi määrittelee *Main*-metodin käyttämällä MSIL:n avainsanaa *.method*. Näemme myös, että metodi on määritelty olemaan sekä *public* että *static*, jotka ovat *Main*-metodin oletusmääreet. Lisäksi voimme nähdä, että metodi on määritelty olemaan ”*managed*”. Tämä on merkittävä rajaus, koska voit kirjoittaa myös C#-koodia, joka on ”unmanaged” tai ”unsafe”. Hallitsemattomasta (unmanaged) koodista C#:ssa puhutaan luvussa 17, ”Yhteistoiminta hallitsemattoman koodin kanssa.”

Seuraava koodirivi käyttää MSIL:n avainsanaa *.entrypoint* määritelläkseen tämän nimenomaisen metodin sovelluksen alkukohdaksi. Kun .NETin ajonaikainen ympäristö käynnistää tämän sovelluksen, siirtyy toiminto juuri tähän pisteeseen.

Seuraavat mielenkiintoiset rivit välillä IL_0000 ja IL_0005 ovat todellista MSIL:n välikoodia. Ensimmäinen käyttää *ldstr* (Load String) välikoodikäskyä ladatakseen pinoon kovakoodatun vakion (”Hello, World”). Seuraava koodirivi kutsuu *System.Console.WriteLine*-metodia. Huomaa, että MSIL lisää metodin nimen eteen sen koosteen nimen, joka määrittelee metodin. Se on tässä MSIL ominaisuudessa hienoa, koska nyt voit entistä helpommin kirjoittaa riippumattoman työkalun, joka käy sovelluksen läpi määritelläkseen, mitä tiedostoja se tarvitsee toimiakseen kunnolla. Voit lisäksi nähdä parametrien määrän (ja niiden tyypit) joita metodi tarvitsee. Tässä tapauksessa *System.Console.WriteLine*-metodi odottaa *System.String*-objektin olevan pinossa, kun sitä kutsutaan. Lopuksi rivillä IL000a on yksinkertainen MSIL välikoodikäsky *ret*, joka merkitsee paluuta funktiosta.

ILDASM on tehokas työkalu. Kun viittaan C#-kääntäjän tuottamaan MSIL:ään, voit käynnistää ILDASMin ja seurata siitä.

> Huomaa Kun haluat selvittää onko EXE tai DLL hallittu, yritä avata se ILDASMiin. Jos tiedosto on kelvollinen hallittu tiedosto, joka sisältää MSIL:n ja luettelon, se avautuu. Jos se ei ole, saat virheilmoituksen, joka kertoo, että *<tiedostosi nimi> has no valid CLR header and cannot be disassembled*.

C# ohjelmointiohjeet

Lopetan tämän kappaleen muutamilla C#-sovelluksen kirjoitusohjeilla.

Milloin määrittelet oman nimiavaruuden

”Hello, World” -sovelluksessa käytimme *Console.WriteLine*-metodia, joka on määritelty *System*-nimiavaruudessa. Itse asiassa kaikki .NETin tyypit ja luokat on määritelty nimiavaruuksissa. Me emme kuitenkaan luoneet nimiavaruutta sovelluksellemme, joten tarkastellaanpa hieman asiaa.

Nimiavaruudet ovat hyvä tapa luokitella tyypit ja luokat niin, että ei synny nimiselkkauksia. Microsoft sijoitti kaikki .NETin luokka- ja tyyppimäärittelyt määrättyyn nimiavaruuteen, koska se halusi varmistaa, että sen nimet eivät sekoitu kenenkään sen kääntäjiä käyttävien nimiin. Mutta kun mietit, pitäisikö sinun käyttää nimiavaruutta, tee itsellesi kysymys: tullaanko luomiani tyyppejä ja luokkia käyttämään ympäristössä, jota en itse hallitse? Toisin sanoen, jos koodiasi käyttää vain oman kehitystiimisi jäsenet, voit helposti muodostaa sellaiset nimeämissäännöt, että väärinkäsityksiä ei synny. Jos kuitenkin kirjoitat luokkia, joita käyttävät monet ulkopuolisetkin ohjelmoijat, jolloin sinä et pysty vaikuttamaan nimipolitiikkaan, sinun tulee ilman muuta käyttää nimiavaruuksia. Lisäksi, koska Microsoft suosittelee yrityksen nimen käyttämistä ylimpänä nimiavaruutena, suosittelen nimiavaruuksien käyttämistä aina, kun joku muu voi nähdä koodisi. Kutsu sitä vaikka ilmaiseksi mainokseksi.

Nimeämisohjeet

Tilastollisesti suurimmat kulut sovelluskehityksessä on aina aiheutunut ylläpidosta. Ennen kuin jatkamme, haluan puhua muutaman minuutin nimeämiskäytännöstä, koska vakaa ja helposti ymmärrettävä nimeämiskäytäntö mahdollistaa helpommin ymmärrettävän ja siten helpommin ylläpidettävän koodin kirjoittamisen.

Kuten useimmat meistä tietävät, nimeämiskäytäntö on henkilökohtainen asia. Asia oli helpompi silloin, kun Visual C++ ja MFC ilmestyivät. Muistan kohdanneeni tämän ongelman, kun oli pääsuunnittelija Peachtree Software -yhtiöllä tiimissä, jonka tehtävänä oli kirjoittaa yhtiön ensimmäinen kirjanpitosovellus MFC:llä. Se oli yksi noista projektin alkukokouksista, jossa jokainen halusi olla valmiina taistelemaan mistä tahansa filosofisesta seikasta vastakohtana niille projektin myöhemmän vaiheen kokouksille, jossa ihmiset halusivat vain saada sen saamarin sovelluksen pois käsistään. Kun ohjelmoijat saapuivat, pystyit erottamaan heidän silmiensä pilkahduksesta, että he olivat valmiita taistelemaan. Tilanteessa, joka oli vaarassa ajautua veriseen kaikki-saavat-vapaat-kädet-ratkaisuun, mitä tein? Osoitin, että koska suuri osa MFC-ohjelmakehitysajasta kului Microsoftin koodin

tutkimiseen, meidän tulisi käyttää samaa nimeämiskäytäntöä, joka Microsoftilla oli MFC:n ohjelmoinnissa. Loppujen lopuksi ei olisi tehokasta käyttää kahta nimeämistapaa järjestelmän lähdekoodissa, toinen MFC:n ja toinen oma. Tietenkin se tosiseikka, että pidän unkarilaisesta merkintätavasta, ei erityisesti haitannut.

Nyt on kuitenkin uusi tilanne ja meillä on C#:ssa uusi kieli ja uudet haasteet. Tässä ympäristössä emme näe Microsoftin koodia. Vieläpä, kun olin keskustellut monen Microsoftin C#-ryhmän jäsenen kanssa, huomasin, että standardi kehittyy. Se saattaa lopuksi olla hieman erilainen kuin mitä tässä esitän, mutta saat ainakin jonkinlaisen lähtökohdan.

Nimeämisstandardit

Ennen kuin selvitän missä ja miten nimeät sovelluksesi eri osat, katsotaan lyhyt yhteenveto nykyään käytössä olevista standardeista.

Unkarilainen merkintätapa

Unkarilainen merkintätapa on järjestelmä, jota käyttävät useimmat C ja C++-ohjelmoijat (myös Microsoftilla). Se on täydellinen nimeämisjärjestelmä, jonka on kehittänyt Microsoftin insinööri Charles Simonyi. Jo 1980-luvun alkupuolella Microsoft otti käyttöön tämän suositun (tai epäsuositun, näkökulmasta riippuen) merkintätavan, joka perustui Simonyin tohtorinväitöskirjaan ”Meta-programming: A Software Production Method”.

Unkarilainen merkintätapa määrittää, että kuhunkin muuttujanimen lisätään etuliite, joka määrittelee sen tyypin. Jokaiselle tyypille ei ole kuitenkaan sovittu tällaista liitettä. Lisäksi, koska uusi kieliä on kehitetty ja uusia tyyppejä luotu, on pitänyt keksiä uusia etuliitteitä. Sen vuoksi, vaikka menet unkarilaista merkintätapaa myyvään kauppaan, saatat kohdata muutaman etuliitteen, joita et odottanut näkeväsi. (Muuten, termi ”Unkarilainen merkintätapa” tulee siitä tosiasiasta, että etuliitteet saavat muuttujanimet näyttämään siltä kuin ne olisi kirjoitettu jollain muulla kielellä kuin englanniksi. Ja Herra Simonyi on unkarilainen.)

Ehkä tärkein julkaisu, joka edisti unkarilaisen merkintätavan käyttöä, oli ensimmäinen kirja, jonka lähes jokainen Windows ja OS/2-ohjelmoija luki: Charles Petzoldin *Programming Windows* (Microsoft Press), joka käytti unkarilaisen merkintätavan murretta omissa esimerkeissään. Lisäksi Microsoft otti merkintätavan käyttöön omassa koodissaan. Kun MFC julkaistiin, sen lähdekoodin mukana tuli joukko uusia, erityisesti C++-koodaukseen sopivia etuliitteitä ja siten merkintätavan käyttö myös jatkossa varmistui.

Joten miten emme yksinkertaisesti jatkaisi unkarilaisen merkintätavan käyttöä? Koska unkarilainen merkintätapa on käyttökelpoinen tilanteissa, jossa käytettävän muuttujan tyyppi, tai näkyvyysalue, on tarpeen tietää. Kuitenkin, kuten tulet näkemään yksityiskohtaisesti seuraavassa luvussa, kaikki C#:n tyypit ovat objekteja ja perustuvat

.NETin System.Object-luokkaan. Tällöin kaikilla muuttujilla on sama perusta toiminnallisuudessa ja käyttäytymisessä. Siksi unkarilaisen merkintätavan käyttötarve .NET-ympäristössä on pienentynyt.

> Huomaa Ihan sivuhuomautuksena niille, jotka kärsivät unettomuudesta: alkuperäinen unkarilaisen merkintätavan esittelevä julkaisu löytyy osoitteesta *http://msdn.microsoft.com/library/techart/hunganotat.htm*.

Pascal-merkintä ja kameli-merkintä

Vaikka on ollut mahdotonta selvittää C#:n ”oikeaa” standardia, on selvää kehitystiimin kirjoitusten perusteella, että he käyttävät merkintää, jonka laittoi alulle Microsoftin työntekijä Rob Caron Hän suositteli Pascal- ja kameli-merkintöjen yhdistelmää muuttujien nimeämisessä. Kirjoituksessaan ”Coding Techniques and Programming Practices,” joka löytyy MSDN:stä (*http://msdn.microsoft.com/library/techart/cfr.htm*), hän suositteli Pascal-merkinnän käyttämistä metodien nimissä, jolloin ensimmäinen kirjain on iso, ja kameli-merkintää muuttujien nimissä. Olen käyttänyt samaa nimeämistapaa tämän kirjan esimerkisovelluksissa. Koska C# kuitenkin sisältää muitakin osia kuin metodit ja muuttujat, olen seuraavissa kappaleissa luetellut eri osat ja nimeämistavat, joita olen nähnyt Microsoftin sisäisesti käyttävän ja kuten olen itsekin päättänyt tehdä.

> Huomaa Lisätietoja tästä asiasta löydät .NET Frameworkin osalta .NET Framework SDK Documentation -julkaisusta, joka löytyy kansiosta .NET Framework Developer Specifications\.NET Framework Design Guidelines\Naming Guidelines.

Nimiavaruudet

Käytä yrityksesi tai tuotteesi nimeä ja käytä isoja ja pieniä kirjaimia kuitenkin niin, että ensimmäinen kirjain on iso, esimerkiksi Microsoft. Jos olet yrityksessä, joka myy ohjelmistoja, tee ylimmän tason nimiavaruus yrityksesi nimestä ja sen alle jokaiselle tuotteella nimiavaruus sen nimen mukaan ja niiden alle tyypit ja muut, jolloin vältät nimiongelmat tuotteidesi kesken. Esimerkki tästä löytyy .NET Framework SDK:sta: *Microsoft.Win32*. Tämä menetelmä johtaa pitkiin nimiin, mutta muista, että koodisi käyttäjien

pitää vain määritellä *using*-määre säästääkseen kirjoittamista. Jos yrityksesi nimi on Trey Research, ja myyt kahta tuotetta, taulukkoa ja tietokantaa, anna nimiavaruuksillesi nimet *TreyResearch.Grid* ja *TreyResearch.Database*.

Luokat

Koska objektien oletetaan olevan eläviä, hengittäviä käsitteitä, joilla on ominaisuuksia, nimeä luokat pronomineilla, jotka kuvaavat luokkaa. Kun luokka on yleinen (eli vähemmän erikoistunut), esimerkiksi tyyppi, joka esittää SQL-merkkijonoa, käytä Pascal-merkintätapaa.

Metodit

Käytä Pascal-merkintätapaa kaikissa metodeissa. Metodien on tarkoitus toimia, ne suorittavat työn. Siksi anna metodiesi nimien kertoa, mitä ne tekevät. Esimerkkeinä vaikkapa metodit *PrintInvoice* ja *OpenDatabase*.

Siinä tapauksessa, että metodia käytetään totuusarvona, lisää metodin nimen eteen liite, joka kertoo, mitä metodi tekee. Jos sinulla on esimerkiksi metodi, joka palauttaa Boolean-tyyppisen arvon sen mukaan, onko työasema lukittu vai ei, anna metodin nimeksi vaikkapa *IsWorkStationLocked*. Jos tällaista metodia kytetään *If*-lauseen osana, sen merkitys on paljon selvempi, kuten seuraavassa näet:

```
if (IsWorkStationLocked) ...
```

Metodin parametrit

Käytä Pascal-merkintätapaa kaikissa parametreissa. Anna niille kuvaavat nimet, jotta, kun käytetään IntelliSenseä, käyttäjä voi välittömästi nähdä parametrin käyttötarkoituksen.

Rajapinnat

Käytä Pascal-merkintätapaa kaikissa rajapinnoissa. Yleensä rajapinnan nimen alkuun laitetaan kirjain I, esimerkiksi *IComparable*. (Tämä on ainoa unkarilaisen merkintätavan tyyppinen käytäntö C#:ssa, jonka tiedän.)

Monet ohjelmoijat käyttävät samoja sääntöjä nimetessään rajapintoja kuin nimetessään luokkia. Näiden kahden välillä on kuitenkin perustavanlaatuinen ero. Luokat edustavat tietojen ja niitä käsittelevien funktioiden kokonaisuutta. Rajapinnat taas edustavat käyttäytymistä. Rajapinnan toteuttamalla sanot, että luokka voi esittää tuon käyttäytymisen. Siksi on yleistä nimetä rajapinnat adjektiiveilla. Esimerkiksi rajapinta, joka määrittelee metodit tietojen serialisointiin (tallentamiseen), voisi olla nimeltään *ISerializable*, joka kertoo käyttäytymisen, jonka rajapinnan toteuttava luokka toteuttaa.

Luokan jäsenet

Tämä on ehkä hankalin kohta C#-ohjelmoijille. Ne meistä, jotka tulevat C++:n ja MFC:n parista, ovat tottuneet aloittamaan jäsenten nimet yhdistelmällä *m_*. Suosittelen kuitenkin kameli-merkintätavan käyttämistä, jossa ensimmäinen kirjain ei ole iso. Tällöin, jos sinulla on metodi, joka tarvitsee *Foo*-nimisen parametrin, voit erottaa sen muuttujan sisäisesta esityksestä luomalla sisäisen jäsenmuuttujan nimeltään *foo*.

On tarpeetonta käyttää jäsenen nimen alussa luokan nimeä. Otetaan esimerkiksi luokka *Author*. Sen sijaan, että tekisit jäsenen nimeltä *AuthorName*, joka tulisi sitten esittää muodossa *Author.AutohorName*, tee yksinkertaisesti jäsen nimeltä *Name*.

Yhteenveto

C#-kielisen ohjelman kirjoittaminen, kääntäminen ja suorittaminen on tärkeä ensimmäinen askel kielen tutkimisessa. Vaikka yleisesti ottaen ei olekaan väliä, millä editorilla teet C#-kieliset lähdetiedostosi, on etua käyttää C#-kehitykseen tehtyjä tehokkaampia editoreja ja ympäristöjä. C#-kääntäjän valintojen ja asetusten tunteminen auttaa ymmärtämään, miten kääntäjä tekee MSIL-koodia. Voit tutkia välikoodia ILDASMin tapaisilla työkalulla. ILDASM sisältyy Microsoft .NET Framework SDK:hon.

C#-ohjelman rakenne tarjoaa joukon ominaisuuksia, jotka on suunniteltu tekemään ohjelmista turvallisempia, helpompia kirjoittaa ja vähemmän virhealttiita. Näihin ominaisuuksiin kuuluvat esimerkiksi nimiavaruudet ja *using*-määre.

Nimeämiskäytännöt, joihin kuuluvat erityiset merkintätavat, tekevät ohjelmista luettavampia ja helpompia ylläpitää.