Osa II

C#-luokkien perusteet

4

Tyyppijärjestelmä

Microsoft .NET Frameworkin ytimessä on yleinen tyyppijärjestelmä nimeltään .NET Common Type System (CTS). Sen lisäksi, että CTS määrittelee kaikki tyypit, se myös tarkkailee sääntöjä, jotka Common Language Runtime (CLR) asettaa sovelluksille, jotka määrittelevät ja käyttävät näitä tyyppejä. Tässä luvussa tutkimme tätä uutta tyyppijärjestelmää, jotta opit mitä tyyppejä C#-ohjelmoijien käytettävissä on ja ymmärrät, mitä seurauksia eri tyyppien käyttämisestä C#-ohjelmissa on. Aloitamme kertomalla, että jokainen ohjelmointielementti C#:ssa on objekti. Sen jälkeen tutkimme, miten .NET jakaa tyypit kahteen luokkaan, arvotyyppeihin ja viittaustyyppeihin. Selvitämme, miten paketointi mahdollistaa täydellisen olioperusteisen tyyppijärjestelmän tehokkaan toiminnan. Lopuksi tutkimme, miten tyyppimuunnokset toimivat C#:ssa ja aloitamme nimiavaruuksien tutkimisen.

Kaikki ovat objekteja

Useimmissa olioperusteisissa kielissä on kaksi erilaista tyyppiä: kielen sisäiset tyypit (primitiiviset tyypit) ja tyypit, joita kielen käyttäjät voivat määritellä (luokat). Kuten voit arvata, primitiiviset tyypit ovat yleensä yksinkertaisia, kuten merkit, merkkijonot ja numerot ja luokkia käytetään monipuolisiin tyyppeihin.

Kahdenlaiset tyypit aiheuttavat paljon ongelmia. Yksi liittyy yhteensopivuuteen. Sanotaan esimerkiksi, että tarvitset int-tyyppien kokoelman perinteisessä järjestelmässä. Sinun pitää luoda erityinen luokka säilyttämään nämä int-tyypiset arvot. Ja jos haluat myös kokoelman double-tyyppejä, sinun pitää tehdä sama double-tyypille. Syy on se, että primitiivisillä tyypeillä ei yleensä ole mitään yhteistä. Ne eivät ole oikeita objekteja, joten ne eivät periydy yleisestä kantaluokasta. Ne ovat enemmänkin ongelmatyyppejä, joita pitää käsitellä yksilöllisesti niiden omien sääntöjen mukaan. Samanlaiseen ongelman törmätään

perinteisessä järjestelmässä, kun haluat määritellä, että metodi voi saada parametrikseen minkä tahansa kielen tukeman tyypin. Koska nämä primitiiviset tyypit ovat epäyhteensopivia, et voi määritellä parametria näin, jos et kirjoita kääreluokkaa kutakin primitiivistä tyyppiä varten.

Onneksi .NET ja C#-maailmassa ei ole enää näin, koska CTS:ssä kaikki ovat objekteja. Eikä pelkästään kaikki ole objekteja, vaan mikä tärkeintä, kaikki objektit periytyvät yhdestä kantaluokasta, joka on määritelty osana CTS:ää. Tämä kantaluokka, *System.Object*, esitellään kappaleessa "Paketointi ja purkaminen."

Arvotyypit ja viittaustyypit

Sellaisen kielen kehittäminen, jossa kaikki ovat objekteja, ei ole uusi ajatus. Tunnetuin yritys on ollut SmallTalk. Suurin haitta siitä, että kaikki ovat objekteja, on ollut kehno suorituskyky. Jos esimerkiksi yrität SmallTalkissa summata kaksi *double*-tyyppistä arvoa, objekteille varataan todellisuudessa muistia keosta. Lienee turha edes mainita, että tilan varaaminen objektille on äärimmäisen tehotonta, kun haluat vain laskea yhteen kaksi lukua.

CTS:n suunnittelijat kohtasivat myös tämän, eli oli tehtävä tyyppijärjestelmä, jossa kaikki ovat objekteja, mutta joka toimii siitä huolimatta tehokkaasti. Heidän ratkaisunsa oli erottaa CTS:n tyypit kahteen luokkaan: *arvotyyppeihin* (value types) ja *viittaustyyppeihin* (reference types). Nämä nimitykset viittaavat siihen, miten muuttujien tilavaraus tehdään ja miten ne toimivat sisäisesti.

Arvotyypit

Kun sinulla on arvotyyppinen muuttuja, sinulla on muuttuja, joka sisältää varsinaisen tiedon. Siten arvotyypin ensimmäinen sääntö on, että se ei voi olla null. Olen alla varannut muistia luomalla CTS:n tyyppiä *System.Int32* olevan C#-muuttujan. Tässä määrittelyssä 32-bittinen tila varataan pinosta.

```
int i = 32;
```

Lisäksi arvon sijoittaminen muuttujaan *i* aiheuttaa sen, että 32-bittinen arvo siirretään tähän varattuun muistialueeseen.

C#:ssa on muutamia arvotyyppejä, esimerkiksi luetellut tyypit, tietueet ja primitiiviset tyypit. Aina kun määrittelet muuttujan, joka on jokin näistä tyypeistä, varaat ko. tyyppiin liitetyn määrän tavuja pinosta ja käsittelet sitä suoraan. Lisäksi, kun välität arvotyypin muuttujan, välität tuon muuttujan todellisen arvon etkä viittausta siihen.

Viittaustyypit

Viittaustyypit ovat osin samanlaisia kuin viittaukset C++:ssa. C#:ssa ne ovat tyyppiturvallisia osoittimia. Tyyppiturvallinen tarkoittaa, että se ei ole pelkkä osoite, joka osoittaa tai ei osoita minne luulet, vaan viittauksen (milloin ei ole *null*) taataan aina osoittavan objektiin, joka on määriteltyä tyyppiä ja jolle on varattu tila keosta. Huomaa, että viittaus voi olla *null*.

Seuraavassa esimerkissä viittaustyypin (*string*) muuttujalle varataan muistia. Nyt muisti varataan keosta ja varaus palauttaa viittauksen.

```
string s = "Hello, World";
```

C#:ssa on muutamia viittaustyyppejä, kuten luokat, taulukot, delegaatit ja rajapinnat. Aina, kun määrittelet muuttujan, joka on jokin näistä tyypeistä, varaat tyyppiin liitetyn määrän tavuja keosta, ja työskentelet tuohon objektiin saamasi viittauksella etkä käsittele suoraan muistipaikan tavujen (kuten arvotyypissä).

Paketointi ja purkaminen

Kysymys kuuluukin nyt näin: "Miten nämä kahden eri luokan tyypit tekevät järjestelmästä tehokkaamman?" Se tapahtuu *paketoinnin* (boxing) avulla. Lyhyesti sanottuna: paketointi on arvotyypin muuntaminen viittaustyypiksi. Päinvastaisessa tapauksessa viittaustyyppi *puretaan* (unboxing) takaisin arvotyypiksi.

Tässä on hienoa se, että objekti on objekti vain silloin, kun sen pitää olla. Sanotaan esimerkiksi, että määrittelet *System.Int32*-tyyppisen muuttujan. Muisti tätä muuttujaa varten varataan pinosta. Voit välittää tämän muuttujan jokaiselle metodille, joka ottaa vastaan *System.Object*-tyypin ja voit käsitellä sen jokaista jäsentä, johon sinulla on oikeus. Siksi se näyttää ja tuntuu aivan objektilta. Oikeasti se kuitenkin on vain neljä tavua pinossa.

Ainoastaan silloin, kun yrität käyttää muuttujaa sen kantaluokan *System.Object* ominaisuuksiin sisältyvällä tavalla, järjestelmä automaattisesti paketoi muuttujan niin, että siitä tulee viittaustyyppi ja sitä voidaan käyttää kuin objektia. Paketoinnin avulla C#:ssa on mahdollista, että kaikki ovat kuin objekteja ja siten voidaan välttää tehottomuus, joka seuraisi siitä, että kaikkien pitäisi olla oikeita objekteja. Katsotaan muutamaa asiaa selventävää esimerkkiä.

```
int foo = 42;          // Arvotyyppi.
object bar = foo;      // foo paketoidaan bar:iin.
```

Koodin ensimmäisellä rivillä luomme *int*-tyyppisen muuttujan *foo*. Kuten tiedät, *int* on arvotyyppi (koska se on primitiivinen tyyppi). Toisella rivillä kääntäjä huomaa, että muuttuja *foo* kopioidaan viittaustyyppiin, jota esittää muuttuja *bar*. Nyt kääntäjä lisää koodin MSIL:ään, joka suorittaa tämän arvon paketoinnin.

bar-muuttuja muunnetaan takaisin arvotyypiksi eksplisiittisellä tyyppimuunnoksella:

```
int foo = 42;          // Arvotyyppi.
object bar = foo;      // foo paketoidaan bar:ksi.
int foo2 = (int)bar;  // puretaan  takaisin int-tyypiksi.
```

Huomaa, että kun paketoidaan, eli muunnetaan arvotyyppi viittaustyypiksi, ei tarvita eksplisiittistä tyyppimuunnosta. Mutta kun puretaan, eli muunnetaan viittaustyyppi arvotyypiksi, tarvitaan tyyppimuunnos. Syy on se, että purkamisessa objekti voidaan muuttaa miksi tahansa tyypiksi. Tällöin eksplisiittinen muunnos on tarpeellinen, jotta kääntäjä voi tarkistaa, että muunnos on kelvollinen. Koska tyyppimuunnokseen liittyy tiukkoja sääntöjä, ja koska näitä sääntäjä valvoo CTS, tarkastelemme tätä tarkemmin tämän luvun kappaleessa "Tyyppimuunnokset tyyppien välillä."

Kaikkien tyyppien äiti: *System.Object*

Kuten aiemmin mainitsin, kaikkien tyyppien on periydyttävä *System.Object*-tyypistä. Siten varmistetaan, että jokaisella järjestelmän tyypillä on määrätty minimijoukko ominaisuuksia. Taulukossa 4-1 kuvataan neljä julkista metodia, jotka kaikki tyypit perivät kantaluokasta.

Taulukko 4-1 *System.Object*-tyypin julkiset (public) metodit

Metodin nimi	Kuvaus
bool Equals()	Tämä metodi vertaa suorituksen aikana kahta objektiviittausta määritellen, ovatko ne täsmälleen samoja objekteja. Jos kaksi muuttujaa viittaa samaan objektiin, on paluuarvo true. Arvotyyppien kohdalla metodi palauttaa truen, jos tyypit ovat samoja ja niillä on sama arvo.
int GetHashCode()	Palauttaa objektille määritellyn hash-avaimen. Hash-funktioita käytetään, kun luokan toteuttaja haluaa sijoittaa luokan hash-avaimen hash-tauluun suorituskyvyn parantamiseksi.
Type GetType()	Käytetään reflection-metodien kanssa (puhutaan tarkemmin luvussa 16, “Metadatan kyseleminen Reflection-metodien avulla”) palauttamaan annetun objektin tyyppitiedot.
string ToString	Oletuksena tätä metodia käytetään palauttamaan objektin nimen. Se voidaan ylikuormittaa palauttamaan käyttäjäystävällisemmän kuvauksen objektista.

Taulukko 4-2 kuvaa *System.Object*-tyypin suojatut metodit.

Taulukko 4-2 *System.Object*-tyypin suojatut (protected) metodit

Metodin nimi	Kuvaus
void Finalize()	Tätä metodia kutsutaan suorituksen aikana suorittamaan puhdistustoimet ennen roskienkeruuta. Huomaa, että tätä metodia ei välttämättä kutsuta. Älä siksi sijoita tähän metodiin sellaista koodia, joka pitää suorittaa. Tämä liittyy termiin *deterministinen lopetus* (deterministic finalization), josta puhutaan tarkemmin luvussa 5, "Luokat."
Object MemberwiseClone	Tämä metodi suorittaa objektin *kevyen kopioinnin* (shallow copy). Tarkoitan tällä sellaista objektin kopiota, johon sisältyy kopioitavan objektin viittaukset muihin objekteihin, mutta ei itse viitattavia objekteja. Jos luokkasi tulee tukea *raskasta kopiointi*a (deep copy), johon sisältyvät myös viitatut objektit, sinun pitää toteuttaa rajapinta *ICloneable* ja tehdä kloonaus tai kopionti ohjelmallisesti.

Tyypit ja peitenimet

CTS on vastuussa niiden tyyppien määrittelystä, joita voidaan käyttää kaikissa .NET-kielissä. Useimmat kielet toteuttavat noille tyypeille peitenimet. Esimerkiksi neljätavuinen kokonaislukuarvo määritellään CTS:n tyypillä *System.Int32*. C# määrittelee tälle peitenimen *int*. Voit käyttää kumpaa haluat, mitään eroa niiden välillä ei ole. Taulukossa 4-3 on lueteltu eri CTS-tyypit ja niiden peitenimet C#:ssa:

Taulukko 4-3 CTS-tyypit ja peitenimet

CTS-tyypin nimi	C# peitenimi	Kuvaus
System.Object	*object*	Kaikkien CTS-tyyppien kantatyyppi.
System.String	*string*	Merkkijono
System.SByte	*sbyte*	Etumerkillinen 8-bittinen arvo
System.Byte	*byte*	Etumerkitän 8-bittinen arvo
System.Int16	*short*	Etumerkillinen 16-bittinen arvo
System.UInt16	*ushort*	Unsigned 16-bittinen arvo
System.Int32	*int*	Etumerkillinen 32-bittinen arvo
System.UInt32	*uint*	Unsigned 32-bittinen arvo
System.Int64	*long*	Etumerkillinen 64-bittinen arvo

(jatkuu)

Taulukko 4-3 *(jatkoa)*

CTS-tyypin nimi	C# peitenimi	Kuvaus
System.UInt64	*ulong*	Etumerkitön 64-bittinen arvo
System.Char	*char*	16-bittinen Unicode-merkki
System.Single	*float*	IEEE 32-bittinen liukuluku
System.Double	*double*	IEEE 64-bittinen liukuluku
System.Boolean	*bool*	Boolean -arvo (*true*/*false*)
System.Decimal	*decimal*	128-bittinen tietotyyppi, jonka tarkkuus on 28 tai 29 numeroa. Käytetään pääasiassa taloudellisissa sovelluksissa, joissa tarvitaan suurta tarkkuutta.

Tyyppien väliset muunnokset

Tässä vaiheessa katsotaan tyyppien tärkeintä puolta: *tyyppimuunnosta* (casting). Oletetaan, että meillä on kantaluokka *Employee* ja siitä periytetty luokka *ContractEmployee*. Tällöin seuraava koodi toimii, koska aina voidaan suorittaa tyyppimuunnos periytyvästä luokasta sen kantaluokkaan (upcast):

```
class Employee { }

class ContractEmployee : Employee { }

class CastExample1
{
    public static void Main ()
    {
        Employee e = new ContractEmployee();
    }
}
```

Seuraava ei kuitenkaan ole sallittu, koska kääntäjä ei pysty tekemään implisiittista tyyppimuunnosta alaspäin (downcast):

```
class Employee { }

class ContractEmployee : Employee { }

class CastExample2
{
```

```
    public static void Main ()
    {
        ContractEmployee ce = new Employee(); // Ei  käänny.
    }
}
```

Syy tähän erilaiseen käyttäytymiseen löytyy luvusta 1, "Olio-ohjelmoinnin perusteet", ja termistä korvattavuus (substitutability). Muista korvattavuus-säännöstä, että periytettyä luokkaa voidaan käyttää sen kantaluokan sijasta. Siksi *ContractEmployee*-tyyppistä objektia voidaan aina käyttää *Employee*-objektin sijalla tai sen korvaajana. Siksi ensimmäinen esimerkki kääntyy.

Et voi kuitenkaan muuntaa *Employee*-objektin tyyppistä objektia *ContractEmployee*-tyyppiseksi, koska ei ole mitään takuita, että se tukee *ContractEmployee*-luokan määrittämää rajapintaa. Siksi tyyppimuunnoksessa "alaspäin" (downcast) käytetään implisiittistä tyyppimuunnosta seuraavasti:

```
class Employee { }

class ContractEmployee : Employee { }

class CastExample3
{
    public static void Main ()
    {
        //Tyyppimuunnos alaspäin epäonnistuu
        ContractEmployee ce = (ContractEmployee)new Employee();
    }
}
```

Mutta mitä tapahtuu, jos valehtelemme ja yritämme juksata CTS:ää muuntamalla eksiplisiittisesti kantaluokka periytyväksi luokaksi seuraavasti:

```
class Employee { }

class ContractEmployee : Employee { }

class CastExample4
{
    public static void Main ()
    {
        Employee e = new Employee();
        ContractEmployee c = (ContractEmployee)e;
    }
}
```

Ohjelma kääntyy mutta sen käynnistäminen generoi ajonaikaisen poikkeuksen. Tässä on huomioitava kaksi seikkaa. Ensinnäkin tuloksena ei saada käännösvirhettä, koska *e* saattaa todellisuudessa olla ylöspäin muunnettu *ContractEmployee*-objekti. Ylöspäin muunnetun objektin todellista tyyppiä ei tiedetä ennen kuin suorituksen aikana. Toiseksi CLR määrittelee objektin tyypin ajon aikana. Kun se tunnistaa kelvottoman tyypin, se aiheuttaa *System.InvalidCastException*-tyyppisen poikkeuksen.

On toinenkin tapa muuntaa objektityyppiä: *as* avainsana. Tämän avainsanan käytöstä on se etu, että jos muunnos on virheellinen, ei CLR aiheuta poikkeusta vaan sijoittaa toiminnon tulokseksi avon *null*. Tässä esimerkki:

```
using System;

class Employee { }

class ContractEmployee : Employee { }

class CastExample5
{
    public static void Main ()
    {
        Employee e = new Employee();
        Console.WriteLine("e = {0}",
                        e == null ? "null" : e.ToString());

        ContractEmployee c  = e as ContractEmployee;
        Console.WriteLine("c = {0}",
                            c == null ? "null" : c.ToString());
    }
}
```

Jos ajat tämän esimerkin, saat seuraavat tulokset:

```
c:>CastExample5
e = Employee
c = null
```

Huomaa, että kyky verrata objektia arvoon *null* tarkoittaa, että sinulla ei ole vaaraa käyttää tyhjää objektia. Itse asiassa, jos esimerkissä olisi yritetty kutsua *c*-objektin *System.Object*-metodia, CTS olisi aiheuttanut *System.NullReferenceException*-poikkeuksen.

Nimiavaruudet

Nimiavaruuksia käytetään määrittämään näkyvyysalue C#-sovelluksissa. Määrittelemällä nimiavaruuden sovelluksen ohjelmoija voi luoda C#-sovellukselle hierarkkisen rakenteen,

joka perustuu toisiinsa liittyvien tyyppien ryhmiin ja muihin sisäkkäisiin nimiavaruuksiin. Useat lähdekooditiedostot voivat kuulua samaan nimiavaruuteen. Jos paketoit useita luokkia samaan nimiavaruuteen, voit määritellä kunkin luokan omassa lähdekooditiedostossaan. Luokkiasi käyttävä ohjelmoija voi saada kaikki nimiavaruuden luokat käyttöönsä *using*-määreen avulla.

> Huomaa On suositeltavaa, aina kuin mahdollista, käyttää yrityksen nimeä ylimmän tason nimiavaruuden nimenä, jotta varmistetaan nimien yksilöllisyys. Katso nimeämisohjeita luvusta 3, "Hello C#".

using-määre

Haluat joskus käyttää määrätyn tyypin täysin määriteltyä nimeä muodossa *nimiavaruus.tyyppi*. Se voi kuitenkin olla melko tylsää eikä aina tarpeellista. Seuraavassa esimerkissä käytän *Console*-objektia, joka sijaitsee *System*-nimiavaruudessa.

```
class Using1
{
    public static void Main()
    {
        System.Console.WriteLine("test");
    }
}
```

Entäpä jos tiedämme, että *Console*-objekti on olemassa vain *System*-nimiavaruudessa? *using*-määreen avulla voimme määritellä nimiavaruuksien etsintäjärjestyksen, jota kääntäjä käyttää törmättyään määrittelemättömään tyyppiin ja lähtiessään hakemaan tyypin määritystä. Seuraavassa esimerkissä kääntäjä löytää *Console*-objektin *System*-nimiavaruudesta ilman, että ohjelmoijan pitää kirjoittaa se joka kerta erikseen:

```
using System;

class Using2
{
    public static void Main()
    {
        Console.WriteLine("test");
    }
}
```

Kun teet todellista sovellusta, jossa on muutama sata kutsua *System*-nimiavaruuden objeteihin, huomaat nopeasti, mitä etua on siitä, että et joudu joka kerta kirjoittamaan objektin täydellistä nimeä.

Et voi määritellä luokan nimeä *using*-määreessä. Siksi seuraava koodi ei ole kelvollinen:

```
using System.Console; // Kelvoton.

class Using3
{
    public static void Main()
    {
        WriteLine("test");
    }
}
```

Sen sijaan voit käyttää seuraavaa *using*-määreen muunnelmaa ja tehdä using-peitenimen:

```
using console = System.Console;

class Using4
{
    public static void Main()
    {
        console.WriteLine("test");
    }
}
```

Tämä on erityisen käyttökelpoinen tilanteessa, jossa sisäkkäiset nimiavaruudet muodostavat pitkän luokan nimen tehden koodista ikävän kirjoittaa ja hankalan lukea.

CTS:n edut

Minkä tahansa ohjelmointikielen tai ajonaikaisen ympäristön yksi tärkeimmista ominaisuuksista on sen tuki tyypeille. Kielelle, jolla on vain rajoitettu määrä tyyppejä tai joka rajoittaa ohjelmoijan mahdollisuuksia laajentaa kielen omia tyyppejä, ei voi odottaa pitkäaikaista menestystä. Yhdistetyllä tyyppijärjestelmällä on kuitenkin muitakin etuja.

Kielten yhteistoiminta

CTS esittää olennaista osaa eri kielten yhteistoiminnassa, koska se määrittelee tyyppijoukon, jota .NET-kääntäjän tulee tukea, jotta yhteistoiminta muitten kielten kanssa toimisi. CTS on määritelty Common Language Specification (CLS):ssä. CLS määrittelee joukon sääntöjä

jokaiselle .NET-kääntäjälle varmistaen, että niiden tuottama koodi toimii CLR:n kanssa johdonmukaisesti. Yksi CLS:n vaatimuksista on, että kääntäjän täytyy tukea määrättyjä CTS:ssä määriteltyjä tyyppejä. Koska kaikki .NET-kääntäjät käyttävät yhtä tyyppijärjestelmää, voit olla varma, että eri kielillä luodut objektit ja tyypit voivat toimia saumattomasti toistensa kanssa. Tämä CTS:n ja CLS:n yhdistelmä tekee kielten yhteistoiminnasta enemmän kuin ohjelmoijan unelman.

Yksikantainen objektihierarkkia

Yksi CTS:n tärkeistä ominaisuuksista on yksikantainen objektihierarkkia. .NET Frameworkissa jokainen järjestelmän tyyppi periytyy kantaluokasta System.Object. Tämä on merkittävä ero C++:aan, jossa ei ole kaikkien luokkien kantaluokkaa. Tämä yhden kantaluokan periaate saa tukea OOP-teoreetikoilta ja se on toteutettu monissa tavallisissa olioperusteisissa kielissä. Yksikantaisen objektihierarkkian edut eivät ole heti havaittavissa, mutta ajan kuluessa alat ihmetellä, miten kielet suunniteltiin ennen

Yksikantainen objektihierarkkia on avain yhdistettyyn tyyppijärjestelmään, koska se takaa, että jokaisella hierarkkian objektilla on yleinen rajapinta ja siksi kaikki hierarkkian objektit ovat pakotetusti samaa kantatyyppiä. Yksi suurimmista haitoista C++:ssa on sen puuttuva tuki tällaiselle hierarkkialle. Katsotaanpa yksinkertaisesta esimerkkiä.

Sanotaan, että rakennat objektihierarkkiaa C++:lla oman kantaluokkasi varaan. Olkoon kantaluokkasi nimi *CFoo*. Sitten haluat yhdistää hierarkkiasi toisen objektihierarkkian kanssa, jonka kaikki luokat periytyvät kantaluokasta *CBar*. Tässä esimerkissä objektihierarkkioilla on erilaiset rajapinnat ja niiden yhdistäminen tulee vaatimaan paljon työtä. Joudut käyttämään jonkinlaisia kääreluokkia tai moniperintää saadaksesi sen toimimaan. Yksikantaisessa objektihierarkkiassa yhteensopivuus ei ole ongelma, koska jokaisella objektilla on sama rajapinta (peritty luokasta *System.Object*). Lopputuloksena tiedät, että jokaisella hierarkkian objektilla (ja mikä parasta, myös kolmannen osapuolen tekemällä .NET-koodilla), on määrätty minimitoiminnallisuus.

Tyyppiturvallisuus

Viimeinen CTS:n etu, jonka tässä yhteydessä mainitsen, on tyyppiturvallisuus. Se takaa, että tyypit ovat sitä, mitä sanovat olevansa ja että määrätylle tyypille voidaan tehdä vain kelvollisia operaatioita. Tyyppiturvallisuus tarjoaa joukon etuja ja ominaisuuksia, kuten

seuraavassa luettelossa kerrotaan. Eduista useimmat ovat yksikantaisen objektihierarkkian ansiota.

- Jokainen viittaus objektiin on tyypitetty ja myös objekti, johon se viittaa, on tyypitetty. CTS takaa, että viittaus osoittaa aina sinne, minne se antaa ymmärtää viittaavansa.
- Koska CTS pitää kirjaa jokaisesta järjestelmän tyypistä, ei ole mahdollista huijata järjestelmää kuvittelemaan, että tyyppi onkin joku muu. Tämä on selvästi tärkeä seikka hajautetuissa järjestelmissä, joissa turvallisuus on etusijalla.
- Kukin tyyppi määrittelee itse jäsentensä käsittelyoikeudet ns. käsittelymääreillä (access modifier). Tämä tehdään jäsenkohtaisesti. Mahdollisia määreitä ovat käsittelyoikeuden antaminen kaikille (määre public), käsittelyoikeuden rajoittaminen vain periytyviin luokkiin (määre protected), käsittelyn estäminen tyypin ulkopuolelta kokonaan (määre private) ja käsittelyoikeuden salliminen vain samassa käännösyksikössä olevien tyypeille (määre internal). Kerron näistä käsittelymääreistä lisää seuraavassa luvussa.

Yhteenveto

Common Type System on .NET Frameworkin tärkeä piirre. CTS määrittelee tyyppijärjestelmän säännöt, joita sovelluksen tulee noudattaa toimiakseen oikein CLR:ssä. CTS:n tyypit on jaettu kahteen luokkaan: viittaustyyppeihin ja arvotyyppeihin. Nimiavaruuksia voidaan käyttää määrittämään sovelluksen näkyvyysalue. Yleisen tyyppijärjestelmän euihin kuuluu kielten välinen yhteistoiminta, yksikantainen objektihierarkkia ja tyyppiturvallisuus. Tyypit voidaan muuntaa C#:ssa paketoinnilla ja purkamisella ja yhteensopivat tyypit voivat jakaa ominaisuuksia ja toiminnallisuutta tyyppimuunnoksen avulla.