5

Luokat

Luokka on jokaisen olioperusteisen kielen sydän. Kuten kerroin luvussa 1, "Olioperusteisen ohjelmoinnin teoria," luokka on tietojen ja niitä käsittelevien metodien kapseloitu yksikkö. Tämä pitää paikkansa jokaisessa olioperusteisessa kielessä. Kielet erottaa toisistaan se, minkä tyyppisiä tietoja voit luokan jäseniin tallentaa ja luokkatyypin ominaisuudet. Luokissa, kuten muissakin ominaisuuksissaan, C# lainaa vähän C++:sta ja Javasta ja lisää omia nerokkaita ratkaisujaan vanhaan ongelmaan.

Tässä luvussa kuvaan ensin luokkien määrittelyn perusasiat C#:ssa, joita ovat instanssijäsenet, käsittelymääreet, muodostimet ja alustusluettelot. Sen jälkeen siirryn määrittelemään staattisia jäseniä ja kerron, mikä ero on vakiolla ja vain-luku kentällä. Sitten käsittelen muodostimet ja termin deterministinen lopetus (deterministic finalization). Luku päättyy lyhyellä selvityksellä periytymisestä ja C#-luokista.

Luokkien määrittäminen

Luokkien määrittelyn syntaksi C#:ssa on yksinkertainen, erityisesti jos yleensä ohjelmoit Javalla tai C++:lla. Sijoitat avainsanan *class* luokkasi nimen eteen ja sen jälkeen luokan jäsenet aaltosulkeiden väliin näin:

```
class Employee
{
    private long employeeId;
}
```

Kuten näet, tämä on perusluokka jos mikä. Meillä on luokka nimeltään *Employee,* joka sisältää yhden jäsenen nimeltään *employeeId*. Huomaa avainsana *private*, joka on jäsenen nimen edessä. Se on nimeltään *käsittelymääre* (access modifier). C#:ssa on neljä eri käsittelymäärettä ja käyn ne läpi aivan kohta.

Luokan jäsenet

Luvussa 4, "Tyyppijärjestelmä," kerroin Common Type System (CTS):n määrittelemistä eri tyypeistä. Nämä kaikki voivat olla C#-luokan jäsenenä. Jäsenet voivat olla:

- **kenttiä** Kenttä (field) on jäsenmuuttuja, jota käytetään arvon säilyttämiseen. OOP-sanastossa kenttiä kutsutaan joskus objektin tiedoiksi tai tietojäseniksi. Voit liittää kenttää useita määreitä sen mukaan, miten haluat niitä käsiteltävän. Näitä määreitä ovat *static*, *readonly* ja *const*. Kerron kohta, mitä nämä määreet merkitsevät ja miten niitä käytetään.
- **metodeja** Metodi (method) on todellinen koodinpätkä, joka käsittelee objektin tietoja (tai kenttiä). Tässä luvussa keskitymme objektin tietoihin, mutta luvussa 6, "Metodit," käsittelemme metodit yksityiskohtaisesti.
- **ominaisuuksia** Ominaisuuksia (property) kutsutaan joskus älykkäiksi kentiksi, koska ne todellisuudessa ovat metodeja, jotka näyttävät luokkaa käyttävän ohjelman näkökulmasta kentiltä. Tämä mahdollistaa korkeamman asteen abstraktion asiakasohjelman kannalta, koska asiakasohjelman ei tarvitse tietää, käsitteleekö se kenttää suoraan vai kutsutaanko käsittelymetodia. Luku 7, "Ominaisuudet, taulukot ja indeksoijat," kertoo ominaisuuksista tarkemmin.
- **vakioita** Kuten nimestä voi päätellä, vakio (constant) on kenttä, jonka arvoa ei voi muuttaa. Puhun myöhemmin tässä luvussa vakioista ja vertaan niitä *vain-luku*-tyyppisiin kenttiin.
- **indeksoijia** Samoin kuin ominaisuus on älykäs kenttä, on indeksoija (indexer) älykäs taulukko, eli jäsen, jonka avulla objekti voidaan indeksoida *get* ja *set* käsittelymetodeilla. Indeksoija mahdollistaa objektin arvojen asettamisen ja hakemisen kätevästi indeksin avulla. Indeksoijista ja ominaisuuksista puhutaan tarkemmin luvussa 7.
- **tapahtumia** Tapahtuma (event) on jotain, joka aiheuttaa tietyn koodin suorittamisen. Tapahtumat ovat oleellinen osa Microsoft Windows-ohjelmointia. Tapahtuma laukeaa esimerkiksi silloin, kun hiirtä liikutetaan, ikkunaan napautetaan tai sen kokoa muutetaan. C#-tapahtumat käyttävät tavallista julkaise/tilaa-menetelmää (publish/subscribe), joka on käytössä Microsoft Message Queuing (MSMQ):ssa ja COM+:n asynkronisessa tapahtumamallissa. Sen avulla sovellus saa asynkronisen tapahtuman käsittelyominaisuudet, mutta C#-mallissa se on kieleen sisällytetty ensisijainen malli. Tapahtumien käytöstä kerrotaan luvussa 14, "Delegaatit ja tapahtumakäsittelijät."

- **operaattoreita** C# antaa sinulle mahdollisuuden operaattorin (operator) ylikuormituksen avulla lisätä luokkiin tavallisia matemaattisia operaattoreita niin, että voit kirjoittaa älykkäämpää koodia. Operaattorin ylikuormitus käydään läpi luvussa 13, "Operaattorin ylikuormitus ja käyttäjän muunnokset."

Käsittelymääreet

Nyt, kun olemme käyneet läpi eri tyypit, joita C#-luokan jäsenenä voi olla, katsotaan miten tärkeää määrettä käytetään määrittämään, miten näkyvä tai käsiteltävissä kukin jäsen on oman luokkansa ulkopuolelta. Näitä määreitä kutsutaan käsittelymääreiksi (access modifier) ja ne on lueteltu taulukossa 5-1.

Taulukko 5-1 C#:n käsittelymääreet

Käsittelymääre	Kuvaus
public	Ilmoittaa, että jäsentä voidaan käsitellä luokan määrittelyn ja periytyvien luokkien hierarkkian ulkopuoltakin.
protected	Jäsen ei ole näkyvissä luokan ulkopuolelle ja sitä voidaan käsitellä vain periytyvissä luokissa.
private	Jäsentä ei voi käsitellä luokan näkyvyysalueen ulkopuolelta. Siten edes periytyvät luokat eivät voi käsitellä tällaista jäsentä.
internal	Jäsen on näkyvissä vain käännösyksikössään. *internal*-käsittelymääre on yhdistelmä määreistä *public* ja *protected*, koska sen täsmällinen merkitys riippuu koodin sijaintipaikasta

Huomaa, että jos et halua jäsenelle oletusmäärettä *private*, sinun tulee määritellä jäsenelle käsittelymääre. Tämä on erilainen käytäntö kuin C++:ssa, jossa sellainen jäsen, jonka käsittelymäärettä ei erikseen kerrottu, sai oletuksena edeltävän jäsenen määreen. Esimerkiksi seuraavassa C++-koodissa jäsenet *a*, *b* ja *c* saavat käsittelymääreen *public* ja jäsenet *d* ja *e* määreen *protected*.

```
class CAccessModsInCpp
{
    public:
    int a;
    int b;
    int c;
    protected:
```

(jatkuu)

```
        int d;
        int e;
}
```

Jotta sama tulos saavutettaisiin C#:lla, pitää koodi muuttaa seuraavanlaiseksi:

```
class AccessModsInCSharp
{
    public int a;
    public int a;
    public int a;
    protected int d;
    protected int d;
}
```

Seuraavassa C#-koodissa jäsen *b* saa käsittelymääreen *private*:

```
public MoreAccessModsInCSharp
{
    public int a;
    int b;
}
```

Main-metodi

Kussakin C#-sovelluksessa tulee olla jossain luokassaan määriteltynä *Main*-metodi. Lisäksi tämä metodi tulee olla määritelty määreillä *public* ja *static*. (Kerron kohta, mitä määre *static* tarkoittaa.) C#-kääntäjän kannalta ei ole merkitystä, missä luokassa *Main*-metodi on, eikä myöskään valitsemasi luokka vaikuta käännösjärjestykseen. Tämä on eri käyttäytyminen kuin C++:ssa, jossa riippuvuudet tulee miettiä tarkkaan, kun sovellus käännetään. C#-kääntäjä on tarpeeksi fiksu käydäkseen lähdekoodisi läpi ja etsien siitä *Main*-metodin itse. Tämä tärkein metodi on kaikkien C#-sovellusten aloituskohta.

Vaikka voit sijoittaa Main-metodin mihin tahansa luokkaan, suosittelen, että teet erillisen luokan pelkästään Main-metodin sijoittamista varten. Tässä esimerkki, joka käyttää tähän vielä yksinkertaista *Employee*-luokkaamme*:*

```
class Employee
{
    private int employeeId;
}
```

```
class AppClass
{
    static public void Main()
    {
        Employee emp = new Employee();
    }
}
```

Kuten näet, tässä esimerkissä on kaksi luokkaa. Tämä on yleinen tilanne C#-ohjelmoinnissa aivan yksinkertaisimmissakin sovelluksissa. Ensimmäinen luokka (*Employee*) on itse ongelmaan liittyvä luokka ja toinen luokka (*AppClass*) sisältää tarvittavan sovelluksen aloituskohdan *(Main)*. Tässä tapauksessa *Main*-metodi instantioi *Employee*-objektin ja jos tämä olisi todellinen sovellus, se käyttäisi *Employee*-objektin jäseniä.

Komentorivin parametrit

Voit käsitellä komentorivin parametreja sovelluksessasi määrittelemällä *Main*-metodi niin, että se ottaa ainoana parametrinaan merkkijonotaulukon. Tällöin voit käsitellä parametreja kuin tavallista taulukkoa. Vaikka taulukot käsitellään vasta luvussa 7, seuraavassa yleinen koodi, joka käy läpi sovelluksen saamat komentoriviparametrit ja tulostaa ne vakiotulostuslaitteella:

```
using System;
class CommandLineApp
{
    public static void Main(string[] args)
    {
        foreach (string arg in args)
        {
            Console.WriteLine(“Argument: {0}", arg);
        }
    }
}
```

Ja tässä esimerkki, jossa kutsun sovellusta muutamalla satunnaiseti valitulla arvolla:

```
e:>CommandLineApp 5 42
Argument: 5
Argument: 42
```

Käskyrivin parametrit annetaan sinulle merkkijonotaulukkona. Jos haluat käsitellä niitä merkkeinä tai valitsimina, sinun pitää ohjelmoida toiminto itse.

> Huomaa Microsoft Visual C++ -ohjelmoijat ovat tottuneet käymään läpi taulukkoa, joka sisältää sovelluksen komentoriviparametrit. Toisin kuin Visual C++:ssa, C#:n komentoriviparametrien taulukko ei kuitenkaan sisällä sovelluksen nimeä taulukon ensimmäisessä alkiossa.

Paluuarvot

Useimmat tämän kirjan esimerkit määrittelevät *Main*-funktion seuraavasti:

```
class SomeClass
{
    §
public static void Main()
{
    §
}
    §
}
```

Voit kuitenkin määritellä *Main*-funktion palauttamaan *int*-tyyppisen arvon. Vaikka se ei olekaan yleinen tapa GUI-sovelluksissa, se voi olla hyödyllinen, kun kirjoitat konsolisovelluksia, jotka on tarkoitettu suoritettavaksi komentojonosta. *return*-käsky lopettaa metodin suorituksen ja paluuarvoa käytetään kutsuvan sovelluksen tai ajotiedoston virhearvona osoittamaan onnistumista tai epäonnistumista. Teet sen seuraavan esimerkin mukaan:

```
public static int Main()
{
§
    // Palauttaa int-tyyppisen arvon.
    // joka kertoo metodin onnistumisen.
    return 0;
}
```

Useita *Main*-metodeja

C#:n suunnittelijat ottivat kieleen mukaan menetelmän, jolla voit määritellä useamman kuin yhden luokan, jossa on *Main*-metodi. Miksi haluaisit tehdä niin? Yksi syy on ohjelman testaaminen. Voit käyttää C#-kääntäjän */main:*<luokan nimi> -valitsinta ja määrätä, minkä luokan Main-metodista ohjelma käynnistetään. Seuraavassa esimerkki, jossa minulla on *Main*-metodi kahdessa luokassa:

```
using System;

class Main1
{
    public static void Main()
    {
        Console.WriteLine(“Main1”);
    }
}

class Main2
{
    public static void Main()
    {
        Console.WriteLine(“Main2”);
    }
}
```

Käännät tämän sovelluksen siten, että *Main1.Main*-metodia käytetään sovelluksen alkukohtana seuraavasti:

```
csc MultipleMain.cs /main:Main1
```

Valitsimen muuttaminen muotoon */main:Main2* käyttää sovelluksen alkukohtana *Main2.Main*-metodia.

Koska C#-kielessä isot ja pienet kirjaimet tulkitaan eri merkeiksi, sinun pitää olla huolellinen luokissa ja valitsimissa. Jos yrität kääntää sellaista sovellusta, jossa Main-metodi on määritelty useassa luokassa ja unohdat määritellä */main*-valitsimen, saat tulokseksi kääntäjän virheilmoituksen.

Muodostimet

Yksi suurimmista eduista C#:n tapaisissa OOP-kielissä on se, että voit määritellä erikoisen metodin, jota kutsutaan aina, kun luokan instanssi luodaan. Tällainen metodi on *muodostin* (constructor). C#:ssa on myös uuden tyyppinen muodostin, nimeltään *staattinen muodostin* (static constructor), jonka tapaat seuraavassa kappaleessa, “Staattiset jäsenet ja instanssijäsenet.”

Muodostimen käytön suurin etu on siinä, että se takaa, että objekti käy läpi kunnollisen alustuksen ennen kuin sitä käytetään. Kun käyttäjä instantioi objektin, sen muodostinta kutsutaan ja siitä pitää palata ennen kuin käyttäjä voi tehdä objektilla mitään. Tämä takuu varmistaa objektin eheyden ja auttaa tekemään olioperusteisella ohjelmointikielellä tehdystä sovelluksesta luotettavamman.

Mutta miten nimeät muodostimen siten, että ajonaikainen ympäristö tietää kutsua sitä, kun instanssi luodaan? C#:n suunnittelijat seurasivat Bjarne Stroustrupia ja määräsivät,

että C#:n muodostimen nimen tulee olla sama kuin itse luokan. Tässä yksinkertainen luokka, jossa on yhtä yksinkertainen muodostin:

```
using System;

class Constructor1App
{
    Constructor1App()
    {
        Console.WriteLine("I'm the constructor.");
    }

    public static void Main()
    {
        Constructor1App app = new Constructor1App();
    }
}
```

Muodostimet eivät palauta arvoja. Jos yrität kirjoittaa muodostimen eteen tyypin, kääntäjä antaa virheilmoituksen, joka sanoo, että et voi määritellä saman nimisiä jäseniä kuin tyyppi on.

Sinun tulee myös huomioida tapa, millä objektit instantioidaan C#:ssa. Se tehdään käyttämällä *new*-avainsanaa seuraavan syntaksin mukaan:

<class> <object> = new *<class>*(*constructor arguments*)

Jos sinulla on C++ tausta, tutki tätä tarkkaan. C++:ssa voit instantioida objektin kahdella tavalla. Voit määritellä sen pinoon näin:

```
// C++ koodi. Tämä luo CMyClass-luokan instanssin pinoon.
CMyClass myClass;
```

Tai voit instantioida objektin vapaaseen muistiin (tai kekoon) käyttämällä C++:n avainsanaa *new*:

```
// C++ koodi. Tämä luo CMyClass-luokan instanssin kekoon.
CMyClass myClass = new CMyClass();
```

Objektin instantiointi on erilainen C#:ssa ja tämä voi aiheuttaa sekaannusta uusille C#-ohjelmoijille. Sekaannus johtuu tosiasiasta, että molemmat kielet käyttävät samaa avainsanaa objektin luomiseen. *new*-avainsanan käyttäminen C++:ssa antaa sinun määritellä minne objekti luodaan, mutta C#:ssa luotavan objektin paikka riippuu instantioitavasta tyypistä. Kuten näimme luvussa 4, viittaustyypit luodaan kekoon ja arvotyypit pinoon. Siksi *new*-avainsana luo sinulle objektin instanssin, mutta se ei määrittele, minne objekti luodaan.

Eli seuraava koodi on kelvollinen C#:ssa, mutta se ei tee sitä, mitä C++-ohjelmoija saattaisi luulla:

```
MyClass myClass;
```

C++:ssa tämä loisi *MyClass*-luokan instanssin pinoon. Kuten mainitsin, voit luoda C#:ssa objekteja vain käyttämällä avainsanaa *new*. Siksi tämä koodirivi määrittelee C#:ssa, että muuttuja *myClass* on tyyppiä *MyClass*, mutta se ei instantioi objektia.

Jos esimerkin vuoksi käännät seuraavan ohjelman, C#-kääntäjä varoittaa sinua määritellystä muuttujasta, jota ei käytetä ohjelmassa:

```
using System;

class Constructor2App
{
    Constructor2App()
    {
        Console.WriteLine("I'm the constructor");
    }

    public static void Main()
    {
        Constructor2App app;
    }
}
```

Siksi, jos määrittelet objektityypin, sinun pitää instantioida se jossain koodissasi käyttämällä *new*-avainsanaa:

```
Constructor2App app;
app = new Constructor2App();
```

Miksi määrittelisit objektin instantioimatta sitä? Objektin määritteleminen ennen sen käyttämistä tehdään tilanteissa, joissa määrittelet luokan toisen luokan sisällä. Tällaisia sisäkkäisiä luokkia sanotaan *sisältymiseksi* (containment) tai *luokkakoosteeksi* (aggregation).

Staattiset jäsenet ja instanssijäsenet

Kuten C++:ssa, voit määritellä luokan jäsenen *staattiseksi jäseneksi* (static member) tai *instanssijäseneksi* (instance member). Oletuksena jokainen jäsen määritellään instanssijäseneksi, joka tarkoittaa sitä, että jäsenen kopio tehdään jokaiseen luokan instanssiin. Kun jäsen määritellään staattiseksi jäseneksi, siitä on olemassa vain yksi kopio.

Staattinen jäsen luodaan, kun luokan sisältävä sovellus ladataan ja se on olemassa koko sovelluksen suoritusajan. Siksi voit käsitellä staattista jäsentä jopa ennen kuin luokasta on luotu instanssia. Mutta miksi tekisit sellaista?

Yksi syy on *Main*-metodi. Common Language Runtime (CLR) tarvitsee aloituskohdan sovellukseesi. Koska CLR:n ei tarvitse luoda jonkin objektisi instanssia, sanoo sääntö, että määrittelet staattisen metodin nimeltä *Main* yhdessä sovelluksesi luokassasi. Haluat myös käyttää staattisia jäseniä, kun sinulla on metodi, joka olioperusteisen ohjelmoinnin periaatteiden mukaan kuuluu semanttisesti johonkin luokkaan, mutta ei tarvitse todellista objektia, esimerkiksi, jos haluat pitää kirjaa siitä, miten monta instanssia määrätystä luokasta on tehty sovelluksen käynnissä olon aikana. Koska staattiset jäsenet ovat olemassa riippumatta objektin instansseista, seuraava koodi toimii:

```
using System;

class InstCount
{
    public InstCount()
    {
    instanceCount++;
    }

    static public int instanceCount = 0;
}

class AppClass
{
    public static void Main()
    {
        Console.WriteLine(InstCount.instanceCount);

        InstCount ic1 = new InstCount();
        Console.WriteLine(InstCount.instanceCount);

        InstCount ic2 = new InstCount();
        Console.WriteLine(InstCount.instanceCount);
    }
}
```

Tämän esimerkin tuloste on seuraava:

```
0
1
2
```

Viimeinen huomautus staattisista jäsenista: staattisella jäsenellä pitää olla kelvollinen arvo. Voit antaa sen samalla, kun määrittelet muuttujan, näin:

```
static public int instanceCount1 = 10;
```

Jos et alusta muuttujaa, CLR tekee sen sovelluksen käynnistyksen yhteydessä käyttämällä oletusarvoa 0. Sen vuoksi seuraavat koodirivit saavat aikaan saman toiminnon:

```
static public int instanceCount2;
static public int instanceCount2 = 0;
```

Muodostimen alustajat

Kaikki C#-objektin muodostimet, lukuunottamatta *System.Object-luokan* muodostinta, suorittavat kutsun kantaluokan muodostimeen juuri ennen muodostimen ensimmäisen rivin suoritusta. Tämän *muodostimen alustajan* (constructor initializer) avulla voit määritellä, minkä luokan mitä muodostinta haluat kutsua. Kutsulla on kaksi mahdollista muotoa:

- *base(…)* -muodon avulla voit kutsua luokan kantaluokan muodostinta. Kutsu tapahtuu kutsun parametrilistan mukaiseen muodostimeen.
- *this(…)*-muodon avulla luokka voi kutsua jotain muuta itsessään määriteltyä muodostinta. Tämä on kätevää silloin, kun olet ylikuormittanut useita muodostimia ja haluat varmistaa, että oletusmuodotinta kutsutaan aina. Metodin ylikuormittamisesta puhutaan luvussa 6, mutta tässä lyhyt määritelmä: Ylikuormitetut metodit ovat saman nimisiä metodeja, joilla on erilainen parametrilista.

Esimerkkinä seuraavan koodin käynnistäminen aiheuttaa luokan A muodostimen suorittamisen ennen luokan B muodostinta.

```
using System;

class A
{
    public A()
    {
        Console.WriteLine(“A”);
    }
}

class B : A
```

(jatkuu)

```
{
    public B()
    {
        Console.WriteLine("B");
    }
}

class DefaultInitializerApp
{
    public static void Main()
    {
        B b = new B();
    }
}
```

Tämä koodi toimii täysin samoin kuin seuraava, jossa kantaluokan muodostinta kutsutaan eksplisiittisesti:

```
using System;

class A
{
    public A()
    {
        Console.WriteLine("A");
    }
}

class B : A
{
    public B() : base()
    {
        Console.WriteLine("B");
    }
}

class BaseDefaultInitializerApp
{
    public static void Main()
    {
        B b = new B();
    }
}
```

Katsotaan parempaa esimerkkiä tilanteesta, jossa muodostimen alustajat ovat hyödyllisiä. Meillä on taas kaksi luokkaa, *A* ja *B*. Tällä kertaa luokalla *A* on kaksi muodostinta, toinen parametriton ja toinen, joka saa parametrikseen int-tyypin muuttujan.

Luokan *B* ainoa muodostin saa parametrikseen myös tyyppiä int olevan muuttujan. Ongelma ilmenee luokan *B* muodostimessa. Jos suoritat seuraavan koodin, kutsutaan luokan *A* muodostimista parametritonta.

```
using System;

class A
{
    public A()
  {
        Console.WriteLine(“A”);
    }

    public A(int foo)
    {
        Console.WriteLine(“A = {0}", foo);
    }
}

class B : A
{
    public B(int foo)
  {
        Console.WriteLine(“B = {0}", foo);
    }
}

class DerivedInitializer1App
{
    public static void Main()
    {
        B b = new B(42);
    }
}
```

Eli, miten voit varmistaa, että haluamaasi *A*-luokan muodostinta kutsutaan? Voit kertoa sen kääntäjälle eksplisiittisesti alustusluettelossa seuraavasti:

```
using System;

class A
{
    public A()
    {
        Console.WriteLine(“A”);
    }
```

(jatkuu)

```
    public A(int foo)
    {
        Console.WriteLine("A = {0}", foo);
    }
}

class B : A
{
    public B(int foo) : base(foo)
    {
        Console.WriteLine("B = {0}", foo);
    }
}

class DerivedInitializer2App
{
    public static void Main()
    {
        B b = new B(42);
    }
}
```

> Huomaa Toisin kuin Visual C++:ssa, et voi käyttää muodostimen alustajaa luokan instanssijäsenten käsittelemiseen.

Vakiot ja vain-luku-tyyppiset kentät

Sinulla tulee takuulla olemaan kenttiä, joiden arvoa ei saa muuttaa ohjelman suorituksen aikana, esimerkiksi sovelluksesi käyttämän tiedoston nimi, piin arvo matemaattisessa ohjelmassa tai yleensäkin mitä tahansa arvoja, jotka eivät muutu ohjelman suorituksen aikana. Näitä tilanteita varten C#:ssa on kaksi hyvin samanlaista tyyppijäsentä: vakio (constant) ja vain-luku-tyyppinen kenttä (read-only field).

Vakiot

Kuten ehkä nimestä arvaatkin, avainsanalla *const* esiteltävät vakiot, ovat kenttiä, joidan arvo säilyy samana koko sovelluksen toiminnan ajan. Kun jotain määritellään vakioksi, on syytä pitää mielessä kaksi asiaa: Ensinnäkin, vakio on jäsen, jonka arvo asetetaan käännöksen aikana, joko ohjelmoijan toimesta tai kääntäjän vakioasetuksella. Ja toiseksi, vakiojäsenen arvo tulee syöttää suoraan, ei minkään lausekkeen tuloksena.

Määrittelet jäsenen vakioksi *const*-avainsanalla seuraavan esimerkin mukaan:

```
using System;

class MagicNumbers
{
    public const double pi = 3.1415;
    public const int answerToAllLifesQuestions = 42;
}

class ConstApp
{
    public static void Main()
    {
        Console.WriteLine("pi = {0}, everything else = {1}",
            MagicNumbers.pi, MagicNumbers.answerToAllLifesQuestions);
    }
}
```

Huomaa tässä yksi seikka. Asiakasohjelman ei tarvitse instantioida *MagicNumbers*-luokkaa, koska oletuksena *const*-jäsenet ovat staattisia. Jotta saat paemman käsityksen asiasta, katso seuraavaa MSIL:n koodipätkää, joka näiden kahden vakion asettamisesta on generoitunut:

```
answerToAllLifesQuestions : public static literal int32 = int32(0x0000002A)
pi : public static literal float64 = float64(3.1415000000000002)
```

Vain-luku-tyyppiset kentät

Vakioksi määritelty kenttä on käyttökelpoinen, koska se kertoo ohjelmoijan tarkoituksesta säilyttää arvo muuttumattomana. Se toimii kuitenkin vain, jos tiedät tuon arvon jo käännöksen aikana. Joten mitä ohjelmoija tekee tilanteissa, joissa kentän arvon tulee pysyä muuttumattomana, mutta arvo tiedetään vasta ohjelman suorituksen aikana? Tätä ei yleensäole muissa kielissä ratkaistu, mutta C#-kielen suunnittelijat tekivät sen *vain-luku-tyyppisen* kentän avulla.

Kun määrittelet kentän *readonly*-avainsanalla, sinulla on mahdollista asettaa sen arvo yhdessä paikassa: muodostimessa. Sen jälkeen sen arvoa ei voi muuttaa luokka itse eikä sitä käyttävä asiakasohjelma. Sanotaan, että haluat graafisessa sovelluksessa pitää tallessa näytön tarkkuustiedot. Et voit tehdä sitä *const*-tyyppisellä kentällä, koska et pysty määrittelemään käyttäjän näytön tarkkuutta ennen kuin ohjelma on käynnissä, joten sinun tulee käyttää seuraavan sivun yläreunassa näkyvää koodia:

```
using System;

class GraphicsPackage
{
    public readonly int ScreenWidth;
    public readonly int ScreenHeight;

    public GraphicsPackage()
    {
        this.ScreenWidth = 1024;
        this.ScreenHeight = 768;
    }
}

class ReadOnlyApp
{
    public static void Main()
    {
        GraphicsPackage graphics = new GraphicsPackage();
        Console.WriteLine("Width = {0}, Height = {1}",
                          graphics.ScreenWidth,
                          graphics.ScreenHeight);
    }
}
```

Ensisilmäyksellä koodi näyttää toimivalta. Siinä on kuitenkin pieni ongelma: määrittelemämme vain-luku-tyyppiset kentät ovat instanssikenttiä, eli käyttäjän tulee instantioida luokka voidakseen käyttää niitä. Tämä ei välttämättä ole ongelma ja se on se, mitä todella haluat tapauksissa, joissa luokan instantiointitapa määrittelee vain-luku-kenttien arvon. Mutta entä jos haluat vakion, joka on staattinen ja joka voidaan alustaa ajon aikana. Tällöin määrittelet kentän määreillä *static* ja *readonly* ja teet *staattisen muodostimen* (static constructor). Se on muodostin, jota käytetään staattisten ja/tai vain-luku-tyyppisten kenttien alustamiseen Seuraavassa olen muokannut edellä ollutta esimerkkiä tehden näytöntarkkuuskentistä staattisia ja vain-luku-tyyppisiä ja olen lisännyt staattisen muodostimen. Huomaa *static*-sanan lisääminen muodostimen määrittelyyn.

```
using System;

class GraphicsPackage
{
    public static readonly int ScreenWidth;
    public static readonly int ScreenHeight;
```

```
    static GraphicsPackage()
    {
        // Code would be here to
        // calculate resolution.
        ScreenWidth = 1024;
        ScreenHeight = 768;
    }
}

class ReadOnlyApp
{
    public static void Main()
    {
        Console.WriteLine("Width = {0}, Height = {1}",
                          GraphicsPackage.ScreenWidth,
                          GraphicsPackage.ScreenHeight);
    }
}
```

Objektin tyhjennys ja resurssien hallinta

Yksi tärkeimmistä komponenttiperusteisen järjestelmän ominaisuuksista on sen kyky suorittaa komponenttien tuhoamisen yhteydessä objektien tyhjennys ja resurssien vapauttaminen. Tyhjennyksellä ja resurssien vapauttamisella tarkoitan muiden komponenttien viittausten oikea-aikaista vapauttamista yhtä hyvin kuin sellaisten niukkojen tai rajallisten resurssien (kuten tietokantayhteydet ja tietoliikenneportit) vapauttamista, joita muutkin käyttävät. Tuhoamisella tarkoitan hetkeä, jolloin objektia ei enää käytetä.

C++:ssa siivous oli selväpiirteinen toiminto, koska se tehtiin objektin hajottimessa eli jokaisessa C++-luokassa määritellyssä funktiossa, joka suoritetaan automaattisesti, kun objekti poistuu näkyvyysalueelta. Microsoft .NET-ympäristössä objektin siivouksesta huolehtii automaattisesti .NETin *roskienkeruu* (carbage collection, GC). Tämä tapa on aiheuttanut kiistaa, sillä verrattuna C++:iin, jossa siivous tapahtuu välittömästi hajottimessa, .NETin ratkaisu perustuu "laiskaan" malliin. GC käyttää taustalla toimivia säikeitä määritellessään, onko objektiin olemassa viittauksia. Toinen GC-säie on sitten vastuussa objektin tuhoamiskoodin suorittamisesta. Tämä on useimmissa tapauksissa järkevä ratkaisu, mutta kaikkea muuta silloin, kun käsitellään resursseja, jotka tulee vapauttaa oikea-aikaisesti ja määrätyssä järjestyksessä. Kuten pelkäät, tämä ei ole helppo ongelma ratkaistavaksi. Tässä kappaleessa käsittelen objektin tuhoamisen ja resurssien hallinnan ongelmaa ja objektien

luontia ennustettavaksi elinajaksi. (Tämä kappale perustuu suurelta osin Microsoftin .NET-tiimin jäsenen Brian Harryn erinomaiseen selvitykseen C#:n resurssien hallinnasta, joka on luettavissa on-line-versiona Microsoftin sivustolla. Kiitokset Brianilla selvityksen käyttöoikeudesta.)

Palanen historiaa

Muutama vuosi sitten, kun .NET-projekti käynnistettiin, raivosi massiivinen väittely resurssien hallinnasta. Ensimmäiset suunnittelijat .NETiin tulivat COM ja Microsoft Visual Basic -ryhmistä, sekä useita muista ryhmistä eri puolilta Microsoftia. Yksi näiden ryhmien kohtaamista suurista ongelmista oli se, miten käsitellä viittauslaskentaan ja sen väärinkäyttöön liittyvät asiat. Yksi näistä ongelmista ovat kehäviittaukset: objekti sisältää viittauksen toiseen, joka taas sisältää viittauksen takaisin ensimmäiseen. Milloin ja miten kehäviittauksessa toisen vapauttaminen voi aiheuttaa ongelman. Yhden tai molempien vapauttamatta jättämnen aiheuttaa muistivuodon, jota on äärimmäisen vaikea selvittää. Ja jokainen, joka on työskennellyt paljon COMin parissa, voi kertoa tarinoita aikataulujen venymisestä, kun tällaisia viittauslaskentaongelmia on selvitelty. Microsoft havaitessaan, että viittauslaskentaongelmat ovat melko yleisiä komponenttiperusteisissa sovelluksissa (kuten COM-sovelluksissa), ryhtyi työstämään yleistä ratkaisua .NETiin.

Alkuperäinen ongelman ratkaisu perustui automaattiseen viittauslaskentaan, joka toimisi, vaikka ohjelmoija unohtaisi käyttää sitä. Laskennan lisäksi ratkaisuun liittyivät metodit, jotka tunnistivat ja käsittelivät kehäviittaukset automaattisesti. Lisäksi .NET-ryhmä suunnitteli lisäävänsä viittauskokoelman, jäljityskokoelman (tracing collection) ja CG-algoritmit, jotka pystyisivät vapauttamaan yhden objektin. Useiden syiden vuoksi, jotka kuvaan kohta, tultiin kuitenkin siihen johtopäätökseen, että tämä ratkaisu ei toimisi kunnolla.

Yksi suurimmista esteistä oli se, että .NET-kehityksen alkuvaiheessa päätavoitteena oli säilyttää mahdollisimman suuri osa Visual Basic -yhteensopivuudesta. Siksi ratkaisun tuli olla täydellinen ja läpinäkyvä ilman merkittäviä muutoksia itse Visual Basic -kieleen. Kiivaan väittelyn jäkeen lopullinen ratkaisu oli ympäristö kontekstiin perustuva malli, jossa kaikki määrätyssä ympäristössä olevat käyttäisivät GC:n päällä viittauslaskuria ja kaikki, jotka eivät olisi tässä ympäristössä, käyttäisivät pelkästään GC:tä. Tämä auttoi välttämään määrätyn tyyppisiä jakautumisongelmia (näistä lisää kohta) mutta ei tuottanut kunnollista ratkaisua kielten välisen koodin hienompaan yhdistämiseen. Joten tehtiin päätös

modernisoida Visual Basicia. Päätöksen osana pudotettiin Visual Basicin yhteensopivuusvaatimuksia. Päätös myös käytännössä johti objektin deterministisen elinkaareen liittyvien seikkojen tutkimseen.

Deterministinen lopetus

Näillä taustatiedoilla määritelkäämme deterministinen lopetus. Kun on selvitetty, että objektia ei enää käytetä, sen lopetuskoodi suoritetaan, joka vapauttaa objektin muihin objekteihin olleet viittaukset. Tämä lopetusprosessi käy luonnollisesti läpi koko objektiketjun, alkaen ylimmän tason objektista. Normaalisti tämä toimii sekä yhteisillä että yhden käyttäjän objekteilla.

Huomaa, että tässä ei puhuta mitään ajasta. Heti, kun GC:n säie huomaa, että viittausta ei enää käytetä, tuo säie ei tee mitään muuta ennen kuin objektin lopetuskoodi on suoritettu. Prosessorin contekstin vaihto voi aina tapahtua käsittelyn aikana, joka merkitsee, että satunnainen (sovelluksen kannalta) aikajakso kuluu ennen kuin tämän vaiheen käsittely päättyy. Kuten edellä mainitsin, on monia tilanteita, joissa sovelluksen kannalta on merkitystä lopetuskoodin suorituksen ajoituksella tai järjestyksellä. Nämä tilanteen liittyvät yleensä resursseihin, joita käyttävät muutkin järjestelmän sovellukset. Seuraavassa muutamia esimerkkejä resursseista, jotka objekti mahdollisesti haluaa vapauttaa heti, kun sitä ei enää tarvita.

- **Muisti** Objektin käyttämän muistin vapauttaminen nopeasti palauttaa sen muiden käyttöön.
- **Ikkunoiden osoittimet (kahvat)** Ikkunaobjektin osoitin GC:ssä ei merkitse todellista kulutusta. Käyttöjärjestelmän sisällä on joitakin ikkunaa esittäviä tietoja ja ikkunaosoittimien määrällä saattaa jopa olla rajoitus, toisin kuin käytettävissä olevalla muistilla.
- **Tietokantayhteydet** Yhtäaikaiset tietokantayhteydet ovat lisensoitu ja siksi niitä saattaa olla käytettävissä vain rajallinen määrä. On tärkeää, että ne vapautetaan uudelleenkäyttöä varten mahdollisimman nopeasti.
- **Tiedostot** Koska määrätystä tiedostosta on olemassa vain yksi instanssi, ja tiedostoa voivat tarvita useat eri toiminnot, sen vapauttaminen heti käyttötarpeen loppuessa on tärkeää.

Viittauslaskentakokoelma

Viittauslaskenta tekee monissa tapauksissa järkevää työtä, kun mahdollistetaan deterministinen lopetus. On kuitenkin syytä huomioida, että kaikissa tapauksissa se ei tee. Yleensä tässä yhteydessä mainitaan kehäviittaukset. Itse asiassa suoraviivainen viittauslaskenta ei ikinä kerää objekteja, jotka osallistuvat kehäviittauksiin. Useimmat meistä ovat epätoivoisesti opiskelleet käsikirjoja pystyäkseen hallitsemaan tämän, mutta näiden tekniikoiden käyttäminen on todellinen virheiden lähde. Joskut väittävät, että heti, kun sinulla on viittaus objektiin, joka sijaitsee kiinteän ohjelman ulkopuolella, olet menettänyt deterministisen lopetuksen, koska sinulla ei ole tietoa siitä, milloin tuo "vieras" koodi vapauttaa viittauksen. Toiset uskovat, että monimutkainen järjestelmä, joka riippuu objektihierarkkian objektien lopetusjärjestyksestä, on luonnostaan helposti vioittuva suunnitelma, joka todennäköisesti aiheuttaa merkittävän ylläpito-ongelman, kun koodia kehitetään.

Jäljityskokoaja

Jäljituskokoaja (tracing collector) ei tunnu yhtä lupaavalta kuin viittauslaskenta. Se on aavistuksen laiskempi ottamaan huomioon lopetuskoodin suorittamisen. Objekteilla on *finalizer*-metodi, joka suoritetaan, kun objekti ei ole enää ohjelma käytettävissä. Jäljityksessä on se pieni etu, että kehäviittauksia ei tarvitse huomioida ja suuri etu, että viittauksen sijoittaminen on yksinkertainen siirto-operaatio (tästä lisää kohta). Tästä maksettava hinta on se, että lopetuskoodin suorittamista ei voi luvata tapahtuvaksi välittömästi sen jälkeen, kun viittausta ei enää käytetä. Entä sitten? Totuus on, että hyvin käyttäytyvässä ohjelmassa (huonosti käyttäytyvä on sellainen, joka kaatuu tai laittaa lopetussäikeen päättymättömään silmukkaan), objektien *finalizer*-metodeja kutsutaan. On-line-dokumentilla on tapana liioitella varoittaessaan lupauksista tässä mielessä, mutta jos objektilla on *finalizer*-metodi, järjestelmä kutsuu sitä. Tämä ei vastaa deterministista lopetusta, mutta on tärkeä ymmärtää, että resurssit kerätään roskiekeruussa ja *finalizer*-metodi on tehokas tapa estää ohjelman resurssien loppuminen.

Suorituskyky

Suorituskyky siinä mielessä kuin se liittyy lopetuksen ongelmaan, on merkittävä seikka. .NET-kehitystiimi uskoo, että jonkinlainen jäljityskokoaja pitää olla käsittelemään kehäviittaukset, joka tekee välttämättömäksi ison osan jäljityskokoajan toiminnasta. Tiimi uskoo myös, että koodin suorituksen suorituskykyyn voidaan pääosin vaikuttaa viittauslaskennan kustannuksella. Hyvä uutinen on se, että kaikkien ajonaikaisen ympäristön tekemien objektien varausten joukossa vain pieni osa tarvitsee determinististä lopetusta. On kuitenkin yleensä vaikea erottaa juuri niiden aiheuttama suorituskyvyn heikkeneminen.

Katsotaanpa yhtä pseudokoodia yksinkertaisesta viittauksen sijoittamisesta, kun käytetään jäljityskokoajaa ja pseudokoodia, kun käytetään viittauslaskentaa.

```
// Jäljitys.
a = b;
```

Se on siinä. Kääntäjä muuntaa rivin yksinkertaiseksi siirto-operaatioksi ja voi jopa joissakin tilanteissa optimoida sen pois kokonaan.

```
// Viittauslaskenta.
if (a != null)
    if (InterlockedDecrement(ref a.m_ref) == 0)
        a.FinalRelease();

if (b != null)
    InterlockedIncrement(ref b.m_ref);

a = b;
```

Tämä koodi kärsii ähkystä, rivejä on enemmän ja kuormitus säädyttömän paljon suurempi, erityisesti koska mukana on kaksi lukituskäskyä. Voit rajoittaa koodin suuruutta sijoittamalla sen ”apumetodiin” ja kasvattaa kutsupolkua. Lisäksi koodin generointi pakostakin kärsii, kun sijoitat kaikki tarpeelliset *try*-lohkot, koska optimoijan kädet ovat käytännössä sidotut poikkeuksenkäsittelykoodin kanssa Tämä pitää paikkansa jopa hallitsemattomassa C++:ssa. Kannattaa myös huomata, että jokainen objekti on neljä tavua suurempi ylimääräisen viittauslaskurikentän takia, joka edelleen kasvattaa muistin käyttöä.

Seuraavat kaksi esimerkkiä näyttävät tämän raskauden, ne on julkaistu .NET-tiimin luvalla. Tämä erityinen testisilmukka varaa tilaa objekteille, tekee kaksi sijoitusta ja tulee ulos yhden viittauksen näkyvyysalueelta. Kuten kaikki testit, tämäkin on avoinna subjektiivisille tulkinnoille. Joku voi sanoa, että tämän rutiinin ympäristössä useimmat viittausten laskennat optimoidaan joka tapauksessa. Se on luultavasti totta. Tässä on kuitenkin yritetty esittää vaikutuksia. Oikeassa ohjelmassa tuollaiset optimoinnit ovat vaikeita, jos eivät peräti mahdottomia, toteuttaa. Itse asiassa C++-ohjelmoijat tekevät tuollaiset optimoinnit käsin, joka johtaa viittauslaskentavirheisiin. Siksi ota huomioon, että oikeissa ohjelmissa sijoitusten suhde on paljon suurempi kuin tässä.

Tässä ensimmäinen esimerkki, ref_gc.cs. Tämä versio luottaa jäljitys GC:hen::

```
using System;

public class Foo
```

(jatkuu)

```
{
    private static Foo m_f;
    private int m_member;

    public static void Main(String[] args)
    {
        int ticks = Environment.TickCount;

        Foo f2 = null;

        for (int i=0; i < 10000000; ++i)
        {
            Foo f = new Foo();

            // Assign static to f2.
            f2 = m_f;

            // Assign f to the static.
            m_f = f;

            // f goes out of scope.
        }

        // Assign f2 to  the static.
        m_f = f2;

        // f2 goes out of scope.

        ticks = Environment.TickCount - ticks;
        Console.WriteLine("Ticks = {0}", ticks);
    }

    public Foo()
    {
    }
}
```

Ja tässä on toinen, ref_rm.cs, viittauslaskuriversio, joka käyttää sisäisiä lukitusoperaatioita säikeessa turvallisuuden vuoksi.

```
using System;
using System.Threading;

public class Foo
{
    private static Foo m_f;
    private int m_member;
    private int m_ref;
```

```
public static void Main(String[] args)
{
    int ticks = Environment.TickCount;

    Foo f2 = null;

    for (int i=0; i < 10000000; ++i)
    {
        Foo f = new Foo();

        // Assign static to f2.
        if (f2 != null)
        {
            if (Interlocked.Decrement(ref f2.m_ref) == 0)
                f2.Dispose();
        }
        if (m_f != null)
            Interlocked.Increment(ref m_f.m_ref);
        f2 = m_f;

        // Assign f to the static.
        if (m_f != null)
        {
            if (Interlocked.Decrement(ref m_f.m_ref) == 0)
                m_f.Dispose();
        }
        if (f != null)
            Interlocked.Increment(ref f.m_ref);
        m_f = f;

        // f goes out of scope.
        if (Interlocked.Decrement(ref f.m_ref) == 0)
            f.Dispose();
    }

    // Assign f2 to  the static.
    if (m_f != null)
    {
        if (Interlocked.Decrement(ref m_f.m_ref) == 0)
            m_f.Dispose();
    }
    if (f2 != null)
        Interlocked.Increment(ref f2.m_ref);
    m_f = f2;

    // f2 goes out of scope.
    if (Interlocked.Decrement(ref f2.m_ref) == 0)
        f2.Dispose();
```

(jatkuu)

```
        ticks = Environment.TickCount - ticks;
        Console.WriteLine("Ticks = {0}", ticks);
    }

    public Foo()
    {
        m_ref = 1;
    }

    public virtual void Dispose()
    {
    }
}
```

Huomaa, että tässä on vain yksi säie ja siksi ei lainkaan väylän sisältöä, joka tekee tästä ihanteellisen tapauksen. Voit ehkä viimeistellä tätä hieman, mutta et paljon. Kannattaa myös huomata, että Visual Basicilla ei ole historiallista pakkoa huolehtia sisäisistä lukitusoperaatioista viittauslaskennassaan (vaikka C++:lla on). Visual Basicin aiemmissa versioissa komponenttia ajettiin yksisäikeisessä osastossa (apartment) ja oli varmaa, että vain yksi säie kerrallaan suoritti sitä. Yksi Visual Basic.NETin tavoitteista oli monisäikeinen ohjelmointi ja toinen päästä eroon COMin monisäikeisestä mallista. Niille teistä, jotka haluavat nähdä version, joka ei käytä lukituksia, seuraa vielä yksi esimerkki, edelleen .NET-tiimin luvalla: ref_rs.cs, viittauslaskuriversio, joka olettaa, että sitä suoritetaan yksisäikeisessä ympäristössä. Tämä ei ole läheskään niin hidas kuin monisäikeinen versio, mutta silti melko paljon hitaampi kuin GC-versio:

```
using System;

public class Foo
{
    private static Foo m_f;
    private int m_member;
    private int m_ref;

    public static void Main(String[] args)
    {
        int ticks = Environment.TickCount;

        Foo f2 = null;

        for (int i=0; i < 10000000; ++i)
        {
            Foo f = new Foo();

            // Assign static to f2.
```

```
            if (f2 != null)
            {
                if (--f2.m_ref == 0)
                    f2.Dispose();
            }
            if (m_f != null)
                ++m_f.m_ref;
            f2 = m_f;

            // Assign f to the static.
            if (m_f != null)
            {
                if (--m_f.m_ref == 0)
                    m_f.Dispose();
            }
            if (f != null)
                ++f.m_ref;
            m_f = f;

            // f goes out of scope.
            if (--f.m_ref == 0)
                f.Dispose();
        }

        // Assign f2 to the static.
        if (m_f != null)
        {
            if (--m_f.m_ref == 0)
                m_f.Dispose();
        }
        if (f2 != null)
            ++f2.m_ref;
        m_f = f2;

        // f2 goes out of scope.
        if (--f2.m_ref == 0)
            f2.Dispose();

        ticks = Environment.TickCount - ticks;
        Console.WriteLine("Ticks = {0}", ticks);
    }

    public Foo()
    {
        m_ref = 1;
    }
```

(jatkuu)

```
    public virtual void Dispose()
    {
    }
}
```

Tässä on ilmiselvästi paljon muuttujia. Silti kaikkien näiden kolmen sovelluksen suorittaminen osoittaa, että GC-versio toimii melkein kaksi kertaa nopeammin kuin yksisäikeinen viittauslaskuriesimerkki ja neljä kertaa nopeammin kuin viittauslaskuri, jossa tehdään lukituksia. Omien kokemusteni keskiarvo, käyttäen IBM ThinkPad 570:tä, jossa oli asennettuna .NET Framework SDK Beta 1, oli seuraava::

```
GC Version (ref_gc)                          1162ms
Ref Counting (ref_rm)(multi-threaded)        4757ms
Ref Counting (ref_rs)(single-threaded)       1913ms
```

Täydellinen ratkaisu

Useimmat lienevät samaa mieltä siitä, että täydellinen ratkaisu tähän ongelmaan saataisiin järjestelmässä, jossa jokainen objekti on nopea instantioida, käyttää ja vapauttaa. Sellaisessa järjestelmässä kukin objekti poistuu deterministisesti säntillisellä tavalla sillä hetkellä, jolloin ohjelmoija uskoo, että objektia ei enää tarvita (riippumatta siitä, onko kehäviittauksia). Käytettyään lukemattoman määrän tunteja tämän ongelman ratkaisemiseen, .NET-tiimi uskoo, että ainoa tapa suorittaa tämä, on yhdistää jäljittävä GC viittauslaskentaan. Suorituskykytesti viittaa, että viittauslaskenta on liian raskas käyttää yleisellä tavalla kaikilla ohjelmointiympäristön objekteilla. Koodit ovat pidempiä ja koodin ja tietojen määrä on suurempi. Jos vielä yhdistät tämän valmiiksi raskaaseen jäljityskerääjään vapauttaaksesi kehäviittaukset, maksat kohtuuttoman hinnan muistin hallinnasta.

Tulee huomioida, että .NET kehityksen alkuaikoina tutkittiin eri tekniikoita, jotta löydettäisiin tapa parantaa viittauslaskennan suorituskykyä. Aiemmin viittauslaskentaan oli rakennettu raskaita järjestelmiä. Valitettavasti kirjallisuuden tarkastelun jälkeen päätettiin, että sellainen järjestelmät suorittivat parantuneen suorituskyvyn luovutamalla osan deterministisyydestään.

Kannattaa ehkä mainita, että C++/COM-ohjelmoijat eivät kärsi tästä ongelmasta. Useat tehokkaat C++-ohjelmat, jotka käyttävät COMia, käyttävät C++ luokkia sisäisesti, joissa ohjelmoijat pitää eksplisiittisesti hallita muistia. C++-ohjelmoijat käyttävät COMia yleensä vain luodessaan rajapintoja asiakasohjelmilleen. Tämä on avainominaisuus, jonka avulla noiden ohjelmien suorituskyky on hyvä. Tämä ei kuitenkaan ole selvästikään .NETin tavoite, josta syystä muistin hallinta on tarkoitus hoitaa GC:llä eikä ohjelmoijan tekemällä koodilla.

(Melkein) täydellinen ratkaisu

Mutta, kuten useimmissa asioissa elämässämme, emme voi saada täydellistä resurssienhallintaakaan. Mutta entä, jos sinulla olisi deterministinen lopetus vain niillä objekteilla, jotka tarvitsevat sitä? .NET-tiimi vietti kauan aikaa miettien tätä. Muista, että tämä oli siinä yhteydessä, joka täsmälleen tuplasi Visual Basic 6:n toiminnot uudessa järjestelmässä. Suurin osa selvityksistä oli vielä käyttökelpoisia, mutta jotkin ideat, jotka hylättiin aikoja sitten, näyttivät nyt entistä miellyttävimmiltä resurssien hallintatekniikoilta, kun yhteensopivuusvaatimus Visual Basic 6:een oli unohdettu.

Ensimmäinen yritys oli yksinkertaisesti merkitä objekti vaatimaan determinististä lopetusta attribuutilla tai periyttämällä se erikoisluokasta. Tämä aiheuttaisi objektin tulon viittauslaskennan piiriin. Monia erilaisia suunnitelmia tutkittiin, esimerkiksi sitä, että molemmat luokat sisältyisivät *System.Object*-luokkaan ja luokkahierarkian juuri muuttuisi johonkin toiseen luokkaan, joka yhdistäisi viittauslaskentamaailman muuhun maailmaan. Valitettavasti oli joukko seikkoja, joita ei voitu kiertää, kuten seuraavissa kappaleissa kerrotaan.

Rakenne

Joka kerta, kun luot objektin, joka tarvitsee deterministisen lopetuksen ja tallennat sen objektiin, joka ei tarvitse, menetät determinismin (koska se ei ole siirrettävä). Ongelma on se, että tämä on vastoin luokkahierarkian ydintä. Miten esimerkiksi taulukot? Jos haluat deterministisen objektien taulukon, pitää taulukon olla deterministinen. Entäpä kokoelmat ja hash-taulut? Luettelo pitenee ja ennen kuin huomaatkaan, koko luokkakirjasto on viittauslaskennassa ja haitat suuremmat kuin edut.

Toinen vaihtoehto oli jakaa .NET Framework luokkakirjasto kahteen haaraan ja tehdä kaksi versiota luokan monista tyypeistä, esimerkiksi deterministinen taulukko ja ei-deterministinen taulukko. Tätä harkittiin vakavasti. Lopullinen päätös kuitenkin oli, että kaksi kopiota koko kirjastosta aiheuttaisi sekaannuksia, suorituskyky olisi huono, koska joka luokasta pitäisi ladata kaksi kopiota eikä se olisi käytännöllistä.

Tutkittiin erikoisratkaisuja erikoisluokkiin mutta mikään niistä ei ollut sellainen, että se olisi voinut skaalata koko kirjastoon. Harkittiin myös, että tehtäisiin perusongelmasta virhetilanne (eli ei-deterministinen objekti sisältää viittauksen deterministiseen), mutta ajateltiin, että tällainen rajoitus tekisi vaikeaksi todellisten ohjelmien tekemisen.

Tyyppimuunnos

Vähän saman tyyppinen ongelma liittyy tyyppimuunnokseen. Mieti seuraavaa: Voinko muuntaa deterministisen objektin *System.Object*-luokan objektiksi? Jos, niin kuuluuko se viittauslaskentaa? Jos vastaus on kyllä, on kaikki viittauslaskennan piirissä. Jos vastaus on ei, objekti menettää determinismin. Jos vastaus on virhe, on unohdettu peruslupaus, että *System.Object* on objektihierarkian juuri.

Rajapinnat

Olet nyt nähnyt, miten monimutkainen tämä asia on. Jos deterministinen objekti toteuttaa rajapintoja, onko rajapintaviittaus-tyyppi viittauslaskennassa? Jos vastaus on kyllä, viittauslaskuri laskee kaikki objektit, jotka toteuttavat rajapintoja. (Huomaa, että *System.Int32* toteuttaa rajapintoja.) Jos vastaus on ei, objekti menettää taas determinisminsä. Jos vastaus on virhe, deterministiset objektit eivät voi toteuttaa rajapintoja. Jos vastaus on ”se riippuu siitä, onko rajapinta merkitty deterministiseksi”, sinulla on toinen haarautumisongelma. Rajapintojen ei oleteta ohjaavan objektin elinkaarta. Mitä jos joku toteuttaa API:n, joka sisältää *ICollection*-rajapinnan ja objektisi, joka toteuttaa sen, tarvitsee determinismin, mutta rajapinta ei ole määritelty siten? Sinulla kävi huono tuuri. Tässä tapauksessa pitäisi määritellä kaksi rajapintaa, toinen deterministinen ja toinen ei, jolloin jokainen metodi tulisi toteuttaa kahdesti. Usko tai ei, mietittiin jopa kahden metodin automaattista generointia. Se tyrmättiin monimutkaisuuden takia.

IDispose-rajapinnan suunnitteluperiaatteet

Joten mihin jäimme? Tällä hetkellä näyttää siltä, että deterministinen lopetus toimii näin:

- GC pitää kirjaa siitä, mihin objekteihin viitataan.
- GC:llä on aina alhaisen prioriteetin säie tutkimassa, milloin sen kirjaamiin objekteihin ei enää viitata.
- Toinen alhaisen prioriteetin säie on vastuussa tyhjennyksestä. Tämä säie kutsuu objektin *Finalize*-metodia.

Tämä ratkaisee kehäviittausongelman, mutta se aiheuttaa muita ongelmia. Ei ole esimerkiksi takuuta, että *Finalize*-metodia kutsutaan ohjelman suorituksen aikana! Tätä tarkoitan sanalla deterministinen, kun puhun deterministisestä lopetuksesta. Kysymys kuuluukin nyt, ”Miten voin tehdä tarpeelliset lopetustoimet ja tietää, että puhdistuskoodiani kutsutaan joka kerta?”

Kun kyseessä on niukkojen resurssien vapauttaminen, Microsoft ehdottaa *Dispose*-menetelmään perustuvaa ratkaisua. Ratkaisu suosittelee, että objekti toteuttaa julkisen metodin, nimeltään esimerkiksi *Cleanup* tai *Dispose*, jota luokan käyttäjää neuvotaan

kutsumaan, kun lopettaa objektin käyttämisen. Tällöin on luokan suunnittelijan asia tehdä tässä metodissa tarpeelliset lopetustoimet. Itse asiassa löydät .NET Framework luokkakirjastosta monia luokkia, jotka toteuttavat *Dispose*-metodin tätä tarkoitusta varten. Esimerkkinä *System.Winforms.TrayIcon*-luokan dokumentti sanoo ”Kutsu *Dispose*-metodia, kun lopetat *TrayIcon*-objektin käyttämisen.”

Periytyminen

Kuten mainitsin luvussa 1, periytymistä käytetään, kun luokka perustuu toiseen luokkaan joko tiedoiltaan tai käyttäytymiseltään, ja se toteuttaa korvattavuuden säännöt, eli että periytyvä luokka voi korvata kantaluokan. Esimerkkinä sopisi tapaus, jossa kirjoitat tietokantaluokkien hierarkkiaa. Sanotaan, että haluat luokan käsittelemään Microsoft SQL Server ja Oracle-tietokantoja. Koska tietokannat ovat määrätyiltä osiltaan erilaisia, sinun pitää tehdä luokka molempia tietokantoja varten. Molemmat tietokannat kuitenkin sisältävät yhteisiäkin toimintoja niin paljon, että haluat sijoittaa ne kantaluokkaan, periyttää molemmat luokat siitä ja korvata tai muokata periytyvien luokkien toiminnot määrätyin osin.

Periytät luokan toisesta seuraavan merkintätavan avulla:

class <*derivedClass*> : <*baseClass*>

Tietokantaratkaisu näyttää esimerkiksi tällaiselta:

```
using System;

class Database
{
    public Database()
    {
        CommonField = 42;
    }

    public int CommonField;

    public void CommonMethod()
    {
        Console.WriteLine(“Database.Common Method”);
    }
}

class SQLServer : Database
{
```

(jatkuu)

```
    public void SomeMethodSpecificToSQLServer()
    {
        Console.WriteLine("SQLServer.SomeMethodSpecificToSQLServer");
    }
}

class Oracle : Database
{
    public void SomeMethodSpecificToOracle()
    {
        Console.WriteLine("Oracle.SomeMethodSpecificToOracle");
    }
}

class InheritanceApp
{
    public static void Main()
    {
        SQLServer sqlserver = new SQLServer();

        sqlserver.SomeMethodSpecificToSQLServer();
        sqlserver.CommonMethod();
        Console.WriteLine("Inherited common field = {0}",
                          sqlserver.CommonField);
    }
}
```

Sovelluksen kääntäminen ja suorittaminen saa aikaan seuraavat tulokset:

```
SQLServer.SomeMethodSpecificToSQLServer
Database.Common Method
Inherited common field = 42
```

Huomaa, että metodit *Database.CommonMethod* ja *Database.CommonField* ovat nyt osa SQLServer-luokan määrittelyä. Koska *SQLServer* ja *Oracle*-luokat periytyvät *Database*-luokasta, ne perivät melkein kaikki sen jäsenet, joiden käsittelymääre on *public*, *protected* tai *internal*. Ainoa poikkeus tähän on muodostin, jota ei voi periä. Jokaisen luokan tulee toteuttaa oma muodostimensa kantaluokasta riippumatta.

Metodin korvaamisesta puhutaan luvussa 6. Selvyyden vuoksi mainitsen kuitenkin tässä vaiheessa, että metodin korvaamisen avulla voit periyttää metodin kantaluokasta ja sen jälkeen muuttaa sen toteutusta. Abstraktit luokat liittyvät tiukasti korvaamiseen ja myös niistä puhutaan tarkemmin luvussa 6.

Monta rajapintaa

Selvennän tätä asiaa omalla kappaleellaan, koska useasta rajapinnasta on tullut kiistelty asia monissa uutisryhmissä ja postituslistoilla. *C# ei tue* useita rajapintoja periytymisen kautta. Voit kuitenkin yhdistää useiden käsitteiden ominaisuudet toteuttamalla useita rajapintoja. Rajapinnat ja miten niitä käytetään, käydään läpi luvussa 9, ”Rajapinnat.” Ajattele C#-rajapintoja nyt kuin ne olisivat COM-rajapintoja.

Seuraava ohjelma on viallinen:

```
class Foo
{
}

class Bar
{
}

class MITest : Foo, Bar
{
    public static void Main ()
    {
    }
}
```

Tästä esimerkistä saamasi virhe liittyy rajapintojen toteuttamiseen. Rajapinnat, jotka haluat toteuttaa, on lueteltu luokan kantaluokan jälkeen. Siksi tässä esimerkissä C#-kääntäjä kuvittelee, että Bar on rajapintatyyppiä ja antaa seuraavan virheilmoituksen::

```
'Bar' : type in interface list is not an interface
```

Seuraava totuudenmukaisempi esimerkki on täysin oikein, koska *MyFancyGrid*-luokka periytyy *Control*-luokasta ja toteuttaa rajapinnat *ISerializable* ja *IDataBound*:

```
class Control
{
}

interface ISerializable
{
}

interface IDataBound
{
}
```

(jatkuu)

```
class MyFancyGrid : Control, ISerializable, IDataBound
{
}
```

Juju tässä on siis se, että ainoa tapa, jolla voit toteuttaa moniperinnän tapaisen asian C#:ssa on käyttää useita rajapintoja.

Sinetöidyt luokat

Jos haluat varmistaa, että luokkaa ei voida koskaan käyttää kantaluokkana, voit käyttää *sealed*-määrettä luokan määrittelyssä. Ainoa rajoitus on se, että abstraktia luokkaa ei voi käyttää sinetöitynä luokkana, koska abstraktin luokan luonne on se, että sitä käytetään kantaluokkana. Toinen seikka tässä on se, että vaikka sinetöidyn luokan tarkoitus on estää tarkoituksettomat periytymisen, määrättyjen suorituksenaikaisten optimointien tuloksena luokka määräytyy sinetöidyksi. Koska kääntäjä takaa, että luokasta ei tule ikinä olemaan periytettyä luokkaa, on mahdollista siirtää virtuaalisen funktiojäsenen toiminnot sinetöidyn luokan instansseista ei-virtuaalisiksi toiminnoiksi. Seuraavassa esimerkki sinetöidystä luokasta:

```
using System;

sealed class MyPoint
{
    public MyPoint(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    private int X;
    public int x
    {
        get
        {
            return this.X;
        }
        set
        {
            this.X = value;
        }
```

```
    }

    private int Y;
    public int y
    {
        get
        {
            return this.Y;
        }
        set
        {
            this.Y = value;
        }
    }
}

class SealedApp
{
    public static void Main()
    {
        MyPoint pt = new MyPoint(6,16);
        Console.WriteLine("x = {0}, y = {1}", pt.x, pt.y);
    }
}
```

Huomaa, että käytän käsittelymäärettä *private* luokan jäsenissä *X* ja *Y*. *protected*-määreen käyttäminen aiheuttaisi kääntäjän varoituksen, koska *protected*-määritellyt jäsenet ovat näkyvissä periytyvissä luokissa, mutta kuten tiedät, sinetöidyllä luokalla ei voi olla yhtään periytyvää luokkaa.

Yhteenveto

Luokan käsite ja sen yhteys objekteihin on olioihin perustuvan ohjelmoinnin perusajatus. C#-kieleen rakennetut olioperusteiset ominaisuudet ovat perintöä C++-kielestä, kuitenkin niin, että niitä on muokattu ja laajennettu .NET Frameworkin ominaisuuksilla. Common Language Runtimen tapaisissa hallituissa järjestelmissä resurssien hallinta on ohjelmoijan jatkuvan mielenkiinnon kohteena. CLR yrittää kovasti vapauttaa ohjelmoijat viittauslaskurien käytön vaivasta käyttämällä deterministiseen lopetukseen perustuvaa roskienkeruuta. Myös periytyminen käsitellään C#:ssa eri tavalla kuin C++:ssa. Vaikka C# tukee vain yksittäisperintää, ohjelmoijat voivat silti saavuttaa joitakin moniperinnän etuja toteuttamalla useita rajapintoja.