Osa III

Koodin kirjoittaminen

10

Lausekkeet ja operaattorit

Tässä luvussa tutkimme jokaisen ohjelmointikielen keskeisintä osaa: sen kykyä suorittaa sijoituksia ja vertailuja operaattorien avulla. Katsomme, mitä operaattorit ovat ja miten niiden suoritusjärjestys määritellään C#:ssa ja sen jälkeen sukellamme lausekkeiden erilaisiin ryhmiin, joiden avulla tehdään esimerkiksi matemaattisia laskutoimituksia, sijoitetaan arvoja ja tehdään vertailuja operandien välillä.

Määritellyt operaattorit

Operaattori on symboli, joka osoittaa yhdelle tai useammalle argumentille tehtävän toiminnon. Toiminnon seurauksena saadaan tulos. Operaattorin kanssa käytettävä merkintäsyntaksi eroaa hieman metodikutsuista, mutta C#:n merkintätapa operaattoreita käyttäville lausekkeilla on tavanomainen. Kuten useiden muidenkin kielten operaattorit, C#-operaattorien käyttötapa seuraa niitä perussääntöjä ja merkintöjä, jotka opimme jo lapsena matematiikan tunneilla. C#:n perusoperaattoreihin kuuluvat kertolasku (*), jakolasku (/), lisäys (+), vähennys (-), jakojäännös (%) ja sijoitus (=).

Operaattorit on tehty tuottamaan uusi arvo lähtöarvojen perusteella. Arvot, joita käsitellään, ovat operandeja. Operaation tulos pitää tallentaa jonnekin muistiin. Joissakin tapauksissa operaation tuottama arvo tallennetaan muuttujaan, joka sisältää yhden alkuperäisen operandin. C#-kääntäjä generoi virheilmoituksen, jos käytät operaattoria ja arvoa ei voida määritellä tai tallentaa. Seuraavassa esimerkissä koodin tuloksena mikään ei muutu. Kääntäjä generoi virheen, koska sellaisen aritmeettisen operaation kirjoittaminen, joka ei muuta vähintään yhtä arvoa, on yleensä ohjelmoijan tekemä virhe.

```
class NoResultApp
{
    public static void Main()
```

```
    {
        int i;
        int j;

        i + j; // Virhe, tulosta ei sijoiteta muuttujaan.
    }
}
```

Useimmat operaattorit toimivat vain numeerisilla tietotyypeillä, joita ovat mm. *Byte*, *Short*, *Long*, *Integer*, *Single*, *Double* ja *Decimal*. Poikkeuksia ovat esimerkiksi vertailuoperaattorit (==“ ja “!=) ja sijoitusoperaattori (=), jotka toimivat myös objekteilla. Lisäksi C# määrittelee + ja += operaattorit *String*-luokalle ja sallii jopa lisäys- ja vähennysoperaattorien (++ ja --) sellaisille kielen rakenteille kuin esimerkiksi delegaatit. Palaan viimeiseen tapaukseen luvussa 14, "Delegaatit ja tapahtumakäsittelijät."

Operaattorien suoritusjärjestys

Kun yksittäinen lauseke tai käsky sisältää useita operaattoreita, kääntäjän pitää määrätä järjestys, jossa nämä operaatiot suoritetaan. Säännöstöä, joka määrää, miten kääntäjä tämän määrittelyn tekee, sanotaan operaattorien arvojärjestykseksi. Operaattorien järjestyksen ymmärtäminen merkitsee kykyä kirjoittaa luotettavia lausekkeita ja helpottaa, kun tutkit yksittäistä riviä ja yrität selvittää, miksi se ei toimi niin kuin pitäisi.

Tutkitaan seuraavaa lauseketta: *42 + 6 * 10*. Jos lasket yhteen 42 ja 6 ja sitten kerrot summan 10, tulos on 480. Jos kerrot 6 kertaa 10 ja lisäät sen tulokseen 42, tulos on 102. Kun koodi käännetään, kääntäjän erikoisosa nimeltä *lexical analyzer* vastaa siitä, miten koodi tulee lukea. Se määrittelee operaattorien suhteellisen järjestyksen, kun lauseke sisältää useita eri operaattoreita. Se tekee sen antamalla arvon kullekin operaattorille. Se operaattori, joka saa suurimman arvon, suoritetaan ensin. Meidän esimerkissämme *-operaattorilla on korkeampi arvo kuin +-operaattorilla, koska * ottaa (tästä kohta lisää) operandinsa ennen kuin +. Syy tähän tulee aritmetiikan perussäännöistä: kerto- ja jakolaskut suoritetaan aina ennen yhteen- ja vähennyslaskuja. Nyt takaisin esimerkkiimme: *:n sanotaan ottavan 6:sen sekä *42 + 6 * 10* lausekkeessa että *42 * 6 + 10* lausekkeessa. siten, että lausekkeet ovat samat kuin *42 + (6 * 10)* ja *(42 * 6) + 10*.

Miten C# määrittelee suoritusjärjestyksen

Katsotaan tarkkaan, miten C# liittää suoritusjärjestyksen operaattoriin. Taulukossa 10-1 on C#:n operaattorien suoritusjärjestys ensimmäisestä viimeiseen. Tämän kappaleen jälkeen käymme tarkemmin läpi erilaisia operaattorityyppejä, joita C# tukee.

Taulukko 10-1 C#-operaattorien suoritusjärjestys

Operaattoriryhmä	Operaattorit
Perusoperaattorit	(x), x.y, f(x), a[x], x++, x--, new, typeof, sizeof, checked, unchecked
Yksioperandiset	+, –, !, ~, ++x, ––x, (T)x
Kerto	*, /, %
Lisäys	+, –
Siirto	<<, >>
Vertailu	<, >, <=, >=, is
Yhtäläisyys	==
Looginen AND	&
Looginen XOR	^
Looginen OR	|
Ehdollinen AND	&&
Ehdollinen OR	||
Ehdollinen	?:
Sijoitus	=, *=, /=, %=, +=, –=, <<=, >>=, &=, ^=, |=

Vasen ja oikea liittyvyys

Liittyvyys määrittelee kumpi puoli lausekkeesta tulee ratkaista ensin. Esimerkkinä seuraava lauseke voi tuottaa tuloksen 21 tai 33 riippuen siitä, kohdistetaanko --operaattoriin vasenta vai oikeaa liittyvyyttä:

```
42-15-6
```

–-operaattorilla on vasen liittyvyys, joka tarkoittaa, että 42-15 ratkaistaan ensin ja sen jälkeen tuloksesta vähennetään 6. Jos --operaattorilla olisi oikea liittyvyys, suoritettaisin ensin operaattorin oikealla puolella oleva lauseke: 15-6 ja sen jälkeen sen tulos vähennettäisiin arvosta 42.

Kaikilla binaarioperaattoreilla, siis sellaisilla, joilla on kaksi operandia, lukuunottamatta sijoitusoperaattoreita, sanotaan olevan vasen liittyvyys eli että lausekkeet ratkaistaan vasemmalta oikealle. Siksi $a + b + c$ on sama kuin $(a + b) + c$, missä $a + b$

ratkaistaan ensin ja summaan lisätään sen jäkeen *c*. Sijoitusoperaattoreilla ja ehdollisilla operaattoreilla sanotaan olevan oikea liittyvyys, eli lausekkeet ratkaistaan oikealta vasemmalle. Toisin sanoen *a = b = c* on sama kuin *a = (b = c)*. Tämä saattaa hämätä joitakin ihmisiä, jotka haluavat kirjoittaa samalle riville useita sjoitusoperaattoreita seuraavan esimerkin tapaan:

```
using System;

class RightAssocApp
{
    public static void Main()
    {
        int a = 1;
        int b = 2;
        int c = 3;

        Console.WriteLine(“a={0} b={1} c={2}", a, b, c);
        a = b = c;
        Console.WriteLine(“After ‘a=b=c’: a={0} b={1} c={2}", a, b, c);
    }
}
```

Tämän esimerkin suorittaminen antaan seuraavat tulokset:

```
a=1 b=2 c=3
After ‘a=b=c’: a=3 b=3 c=3
```

Lausekkeiden ratkaiseminen oikealta saattaa ensin aiheuttaa sekaannusta, mutta ajattele asiaa näin: Jos sijoitusoperaattorilla olisi vasen liittyvyys, kääntäjä ratkaisisi ensin sijoituksen *a = b*, joka antaise muuttujalla *a* arvon 2, ja sitten kääntäjä ratkaisisi sijoituksen *b = c*, jolloin *b* saisi arvon 3. Lopputulos olisi *a=2 b=3 c=3*. Turha lienee mainita, että se ei voi olla odotettu tulos sijoituksista *a = b = c* ja tämä on se syy, miksi sijoitusoperaattoreilla ja ehdollisilla operaattoreilla on oikea liittyvyys.

Käytännön suoritusjärjestyksestä

Mikään ei ole turhauttavampaa kuin etsiä kauan aikaa virhettä, jonka syyksi lopulta selviää se, että ohjelmoija ei ole tietänyt suoritusjärjestykseen tai liitettävyyteen sovellettavia sääntöjä. Olen nähnyt useita postituslistoja, joissa älykkäät ihmiset ovat ehdottaneet ohjelmointiin merkintätätapaa, jossa välilyöntejä käytettäisiin osoittamaan operaattoria, joka heidän mielestää pitäisi suorittaa ensin eli käytettäisiin eräänlaista itsensäkuvaavaa menetelmää. Esimerkkinä, koska kaikki tiedämme, että kertolaskut suoritetaan ennen yhteenlaskua, voisimme kirjoittaa seuraavanlaisen koodin, jossa välilyönti osoittaa halutun suoritusjärjestyksen:

```
a = b*c + d;
```

Tällä menetelmällä on useita puutteita. Kääntäjä ei jäsentele koodia oikein, jos sen rakenne ei ole sovitun mukainen Kääntäjä jäsentää lausekkeet niiden sääntäjen mukaan, jotka kääntäjän tehneet ihmiset ovat laatineet. Sitäpaitsi on olemassa symboli, jonka avulla voi määrätä suoritusjärjestyksen tai liitettävyyden: kaarisulku. Voit esimerkiksi kirjoittaa lausekkeen *a = b * c + d* uudelleen joko näin: *a = (b * c) + d* tai näin: *a = b * (c + d)*, ja kääntäjä suorittaa sulkujen sisällä olevat laskut ensin. Jos lausekkeessa on kaksi tai useampia pareja sulkuja, kääntäjä ratkaisee kunkin sulun sisällön ja sen jälkeen koko lausekkeen käyttämällä edellä kuvattuja suoritusjärjestys- ja liitettävyyssääntöjä.

Vankka mielipiteeni on, että sinun kannattaa käyttää aina sulkuja, kun yhdistät useita operaattoreita samaan lausekkeeseen. Suosittelen tätä vaikka tietäisitkin suoritusjärjestyksen, koska ne, jotka joskus tulevat ylläpitämään koodiasi, eivät välttämättä sitä tiedä.

C#:n operaattorit

On hyödyllistä ajatella operaattoreita niiden suoritusjärjestykseen perusteella. Tässä kappaleessa käyn läpi yleisimmin käytetyt operaattorit siinä järjestyksessä kuin ne aiemmin olivat talukossa 10-1.

Lausekkeen perusoperaattorit

Ensimmäinen operaattoriluokka, jota tutkimme, on lausekkeen perusoperaattorit. Koska monet näistä ovat itsestään selviä, vain luettelen ne ja kerron lyhyesti niiden toiminnon. Sen jälkeen käyn oudommat operaattorit läpi tarkemmin.

- **(x)** Tätä sulku-operaattorin muotoa käytetään ohjaamaan suoritusjärjestystä matemaattisissa operaatioissa tai metodikutsuissa.
- **x.y** "Piste"-operaattoria käytetään luokan tai tietueen jäsenen määrittämiseen. Esimerkissä *x* esittää itse käsitettä (luokka, tietue) ja *y* esittää sen jäsentä.
- **f(x)** Tätä sulku-operaattorin muotoa käytetään luettelemaan metodin parametrit.
- **a[x]** Hakasulkuja käytetään taulukon indekseissä. Niitä käytetään myös yhdessä indeksoijien kanssa, jolloin objekteja voidaan käsitellä kuin taulukkoa. Indeksoijista kerrotaan lisää luvussa 7, "Ominaisuudet, taulukot ja indeksoijat."

- **x++** Koska tästä on paljon puhuttavaa, käsittelen lisäys-operaattorin (++) myöhemmin tässä luvussa kappaleessa "Lisäysoperaattorit ja vähennysoperaattorit."
- **x––** Myös vähennysoperaattorista kerrotaan lisää myöhemmin tässä luvussa.
- **new** new-operaattoria käytetään instantioimaan objekteja luokasta.

typeof

Reflection on kyky hakea suorituksen aikana tietoja tyypistä. Tietoja voivat olla esimerkiksi tyypin nimi, luokkien nimet ja tietueen jäsenet. Tämän kyvyn keskeistä osaa .NETissä esittää luokka nimeltä *System.Type*. Se on kaikkien *reflection*-toimintojen ydin ja se voidaan hakea käyttämällä *typeof*-operaattoria. Emme mene *reflection*-toimintoon tässä yhteydessä pidemmälle, sillä sille on varattu luku 16, "Metadatan kyseleminen Reflection-metodien avulla," mutta tässä kuitenkin yksinkertainen esimerkki, joka näyttää, miten helppoa on käyttää *typeof*-operaattoria lähes millaisten tahansa tietojen hakemiseen tyypistä tai objektista suorituksen aikana:

```
using System;
using System.Reflection;

public class Apple
{
    public int nSeeds;
    public void Ripen()
    {
    }
}

public class TypeOfApp
{
    public static void Main()
    {
        Type t = typeof(Apple);
        string className = t.ToString();

        Console.WriteLine("\nInformation about the class {0}",
            className);

        Console.WriteLine("\n{0} methods", className);
        Console.WriteLine("----------------------------");
        MethodInfo[] methods = t.GetMethods();
        foreach (MethodInfo method in methods)
        {
            Console.WriteLine(method.ToString());
        }

        Console.WriteLine("\nAll {0} members", className);
```

```
        Console.WriteLine("----------------------------");
        MemberInfo[] allMembers = t.GetMembers();
        foreach (MemberInfo member in allMembers)
        {
            Console.WriteLine(member.ToString());
        }
    }
}
```

Ohjelma sisältää *Apple*-nimisen luokan, jolla on vain kaksi jäsentä: kenttä nimeltä *nSeeds* ja metodi nimeltä *Ripen*. Käytän ensin *typeof*-operaattoria ja luokan nimeä hakeakseni *System.Type*-objektin, jonka tallennan muuttujaan *t*. Sen jälkeen voin käyttää *System.Type* -objektin metodeja hakeakseni kaikki *Apple*-luokan metodit ja jäsenet. Tämä tehdään käyttämällä *GetMethods* ja *GetMembers* -metodeja. Lopuksi näiden metodien tulokset on tulostettu oletustulostuslaitteelle ja ne näyttävät tältä:

```
Information about the class Apple

Apple methods
----------------------------
Int32 GetHashCode()
Boolean Equals(System.Object)
System.String ToString()
Void Ripen()
System.Type GetType()

All Apple members
----------------------------
Int32 nSeeds
Int32 GetHashCode()
Boolean Equals(System.Object)
System.String ToString()
Void Ripen()
System.Type GetType()
Void .ctor()
```

Ennen kuin jatkamme eteenpäin, haluan näyttää vielä pari seikkaa. Huomaa ensinnäkin, että luokan perityt jäsenet ovat myös luettelossa. Koska en eksplisiittisesti periyttänyt luokkaa toisesta, tiedämme, että ne jäsenet, joita *Apple*-luokassa ei ole määritelty ovat periytyneet paruskantaluokasta *System.Object*. Toiseksi huomaa, että voit käyttää myös *GetType*-metodia hakeaksesi *System.Type*-objektin. Se on *System.Object*-luokasta peritty metodi, jota voit käyttää objektien kanssa mutta et luokkien. Seuraavan sivun ylälaidassa olevat koodipätkät ovat vaihtoehtoisia; niillä voidaan molemmilla hakea *System.Type*-objekti.

```
// Haetaan System.Type-objekti luokan määrittelystä.
Type t1 = typeof(Apple);


// Haetaan System.Type-objekti objektin perusteella.
Apple apple = new Apple();
Type t2 = apple.GetType();
```

sizeof

sizeof-operaattoria käytetään annetun tyypin koon (tavuina) selvittämiseen. Pidä mielessä kaksi äärimmäisen tärkeää seikkaa, kun käytätä tätä operaattoria. Ensinnäkin, voit käyttää *sizeof*-operaattoria vain arvotyypeillä. Vaikka operaattoria voidaan käyttää luokan jäseniin, sitä ei voida käyttää itse luokkaan. Toiseksi, tätä operaattoria voidaan käyttää vain metodissa tai koodilohkossa, joka on merkitty turvattomaksi (unsafe). Puhumme turvattomasta koodista luvussa 17, "Yhteistoiminta hallitsemattoman koodin kanssa." Tässä esimerkki *sizeof*-operaattorin käytöstä luokan metodissa, joka on merkitty määreellä *unsafe*:

```
using System;

class BasicTypes
{
    // HUOMAA: Sinun pitää määritellä sizeof-operaattoria
    // käyttävä koodi määreellä unsafe.
    static unsafe public void ShowSizes()
    {
        Console.WriteLine("\nBasic type sizes");
        Console.WriteLine("sizeof short = {0}", sizeof(short));
        Console.WriteLine("sizeof int = {0}", sizeof(int));
        Console.WriteLine("sizeof long = {0}", sizeof(long));
        Console.WriteLine("sizeof bool = {0}", sizeof(bool));
    }
}

class Unsafe1App
{
    unsafe public static void Main()
    {
        BasicTypes.ShowSizes();
    }
}
```

Tämän sovelluksen käynnistäminen tuottaa seuraavat tulokset:

```
Basic type sizes
sizeof short = 2
sizeof int = 4
sizeof long = 8
sizeof bool = 1
```

Yksinkertaisten järjestelmän sisäisten tyyppien lisäksi *sizeof*-operaattoria voi käyttää määrittelemään myös käyttäjän luomien arvotyyppien, kuten tietueiden, koko. *sizeof*-operaattorin tulokset saattavat kuitenkin joskus olla yllättäviä, kuten seuraavasta esimerkistä huomaat:

```
// Using the sizeof operator.
using System;

struct StructWithNoMembers
{
}

struct StructWithMembers
{
    short s;
    int i;
    long l;
    bool b;
}

struct CompositeStruct
{
    StructWithNoMembers a;
    StructWithMembers b;

    StructWithNoMembers c;
}

class UnSafe2App
{
   unsafe public static void Main()
   {
    Console.WriteLine("\nsizeof StructWithNoMembers structure = {0}",
                      sizeof(StructWithNoMembers));
    Console.WriteLine("\nsizeof StructWithMembers structure = {0}",
                      sizeof(StructWithMembers));
    Console.WriteLine("\nsizeof CompositeStruct structure = {0}",
                      sizeof(CompositeStruct));
   }
}
```

Odotit varmaan sovelluksen tulostavan arvon 0 tietueelle, jolla ei ole lainkaan jäseniä (*StructWithNoMembers*), arvon 15 tietueelle, jolla on neljä perusjäsentä (*StructWithMembers*) ja arvon 15 tietueelle, joka yhdistää kaksi muuta (*CompositeStruct*), mutta todelliset tulokset ovatkin seuraavat:

```
sizeof StructWithNoMembers structure = 1
```

```
sizeof StructWithMembers structure = 16

sizeof CompositeStruct structure = 24
```

Syy tähän on pakkauksessa ja tietueen sijoittamisessa, jotka liittyvät siihen, miten kääntäjä sijoittelee *struct*-tyypin muuttujan muistiin. Jos *struct* olisi kolme tavua pitkä ja tavun tasaus (byte alignment) olisi asetettu neljää tavuun, kääntäjä automaattisesti pakkaisi yhden ylimääräsen tavun mukaan ja *sizeof*-operaattori ilmoittaisi, että *struct* on neljä tavua pitkä. Sinun tulee ottaa tämä huomioon, kun määrittelet tietueiden kokoa C#:ssa.

checked ja *unchecked*

Nämä kaksi operaattoria ohjaavat ylivuodon tarkistusta matamaattisissa operaatioissa. Koska nämä liittyvät kiinteästi virheenkäsittelyyn, ne käsitellään luvussa 12, "Virheenkäsittely poikkeusten avulla."

Matemaattiset operaattorit

C# tukee samoja matemaattisia perusoperaattoreita kuin lähes kaikki muutkin kielet: kertolasku (*), jakolasku (/), lisäys (+), vähennys (-) ja jakojäännös (%). Neljä ensimmäistä ovat tarkoitukseltaan täysin selviä. Jakojäännös-operaattori tuottaa kokonaislukujen jaon jakojäännöksen. Seuraava koodi esittää näitä matemaattisia operaattoreita toiminnassa:

```
using System;

class MathOpsApp
{
    public static void Main()
    {
        // System.Random-luokka on osa .NET Framework
        // luokkakirjastoa. Sen oletusmuodostin syöttää
        // Next-metodia käyttäen pohjana nykyhetkeä.
        Random rand = new Random();
        int a, b, c;

        a = rand.Next() % 100; // Limit max to 99.
        b = rand.Next() % 100; // Limit max to 99.

        Console.WriteLine("a={0} b={1}", a, b);

        c = a * b;
        Console.WriteLine("a * b = {0}", c);

        // Huomaa, että seuraava koodi käyttää konaislukuja.
```

```
        // Siksi, jos a on pienempi  b, tulos on aina 0.
        // Jos haluat tarkemman tuloksen, käytä muuttujia,
        // joiden tyyppi on double tai float.
        c = a / b;
        Console.WriteLine("a / b = {0}", c);

        c = a + b;
        Console.WriteLine("a + b = {0}", c);

        c = a - b;
        Console.WriteLine("a - b = {0}", c);

        c = a % b;
        Console.WriteLine("a % b = {0}", c);
    }
}
```

Yksioperandiset operaattorit

Yksioperandisia operaattoreita on kaksi, plus ja minus. Yksioperandimen minus osoittaa kääntäjälle, että operandista tulee palauttaa negatiivinen arvo. Siksi seuraavan koodin tuloksena muuttujan *a* arvo on *-42*:

```
using System;

using System;

class Unary1App
{
    public static void Main()
    {
        int a = 0;

        a = -42;
        Console.WriteLine("{0}", a);
    }
}
```

Seuraava koodi aiheuttaa kuitenkin epätietoisuutta:

```
using System;

class Unary2App
{
    public static void Main()
    {
        int a;
        int b = 2;
```

(jatkuu)

```
        int c = 42;

        a = b * -c;
        Console.WriteLine("{0}", a);
    }
}
```

*a = b * -c* voi olla hieman hämäävä. Vielä kerran, käytä sulkuja niin saat koodistasi paljon selkeämpää.

```
// Sulkujen avulla on selvää, että kerrot b:llä
// c:n käänteisarvon.
a = b * (-c);
```

Jos yksioperandinen minus palauttaa operandin negatiivisen arvon, voit kuvitella, että yksioperandinen plus palauttaa operandin positiivisen arvon. Se ei kuitenkaan tee muuta kuin palauttaa alkuperäisen operandin eli sillä ei ole operandiin vaikutusta. Esimerkiksi seuraava koodi saa aikaan tulosteen *-84*.

```
using System;

class Unary3App
{
    public static void Main()
    {
        int a;
        int b = 2;
        int c = -42;

        a = b * (+c);
        Console.WriteLine("{0}", a);
    }
}
```

Jos haluat operandin positiivisen arvon (itseisarvon) käytä *Math.Abs*-funktiota. Seuraava tuottaa tuloksen *84*:

```
using System;

class Unary4App
{
    public static void Main()
    {
        int a;
        int b = 2;
        int c = -42;

        a = b * Math.Abs(c);
        Console.WriteLine("{0}", a);
```

```
    }
}
```

Viimeinen yksioperandinen operaattori, jonka lyhyesti kuvaan, on T(x)-operaattori. Tämä on sulkujen käyttömuoto, jonka avulla voit suorittaa tyyppimuunnoksen (cast) tyypistä toiseen. Koska tämä operaattori voidaan ylikuormittaa käyttäjän määrittelemän muunnoksen suorittamiseksi, se käsitellään luvussa 13, "Operaattorin ylikuormitus ja käyttäjän muunnokset."

Yhdistetyt sijoitusoperaattorit

Yhdistetty sijoitusoperaattori (compound assignment operator) on binaarioperaattorin ja sijoitusoperaattorin (=) yhdistelmä. Sen syntaksi on

rvalue *op*= lvalue

missä *op* tarkoittaa operaattoria. Sen sijaan, että korvaisi arvon *rvalue* arvolla *lvalue*, yhdistetty sijoitusoperaattori tekee seuraavan toiminnon:

rvalue = rvalue *op* lvalue

käyttäen *lvalue*-arvoa lopputuloksen laskennan pohjana.
Huomaa, että käytin sanoja "tekee seuraavan toiminnon." Kääntäjä ei sanatarkasti ottaen tee mitään käännöstä muodosta $x \mathrel{+}= 5$ muotoon $x = x + 5$, se vain loogisesti toimii niin. Itse asiassa operaattorin käyttöön liittyy varoitus silloin, kun *lvalue* on metodi. Tutkitaanpa seuraavaa:

```
using System;

class CompoundAssignment1App
{
    protected int[] elements;
    public int[] GetArrayElement()
    {
        return elements;
    }

    CompoundAssignment1App()
    {
        elements = new int[1];
        elements[0] = 42;
    }

    public static void Main()
    {
        CompoundAssignment1App app = new CompoundAssignment1App();
```

(jatkuu)

```
        Console.WriteLine("{0}", app.GetArrayElement()[0]);
        app.GetArrayElement()[0] = app.GetArrayElement()[0] + 5;
        Console.WriteLine("{0}", app.GetArrayElement()[0]);
    }
}
```

Huomaa lihavoitu rivi eli *CompoundAssignment1App.GetArrayElements*-metodin kutsu ja sen ensimmäisen elementin muokkaus, jossa käytin seuraava syntaksia:

x = x *op* y

Tässä generoitunut MSIL-koodi:

```
// x = x op y:n tehoton tekniikka.

.method public hidebysig static void Main() il managed
{
  .entrypoint
  // Code size       79 (0x4f)
  .maxstack  4
  .locals (class CompoundAssignment1App V_0)
  IL_0000:  newobj     instance void CompoundAssignment1App::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldstr      "{0}"
  IL_000b:  ldloc.0
  IL_000c:  call       instance int32[] CompoundAssignment1App::GetArrayElement()
  IL_0011:  ldc.i4.0
  IL_0012:  ldelema    ['mscorlib']System.Int32
  IL_0017:  box        ['mscorlib']System.Int32
  IL_001c:  call       void ['mscorlib']System.Console::WriteLine
(class System.String,
 class System.Object)
  IL_0021:  ldloc.0
  IL_0022:  call
instance int32[] CompoundAssignment1App::GetArrayElement()
  IL_0027:  ldc.i4.0
  IL_0028:  ldloc.0
  IL_0029:  call
instance int32[] CompoundAssignment1App::GetArrayElement()
  IL_002e:  ldc.i4.0
  IL_002f:  ldelem.i4
  IL_0030:  ldc.i4.5
  IL_0031:  add
  IL_0032:  stelem.i4
  IL_0033:  ldstr      "{0}"
```

```
  IL_0038:  ldloc.0
  IL_0039:  call
instance int32[] CompoundAssignment1App::GetArrayElement()
  IL_003e:  ldc.i4.0
  IL_003f:  ldelema    ['mscorlib']System.Int32
  IL_0044:  box        ['mscorlib']System.Int32
  IL_0049:  call  void ['mscorlib']System.Console::WriteLine
                                (class System.String, class System.Object)
  IL_004e:  ret
} // end of method. 'CompoundAssignment1App::Main'
```

Katsomalla lihavoituja rivejä MSIL-koodissa, näet, että *CompoundAssignment1App.GetArrayElements*-metodia kutsutaan todellisuudessa kahdesti! Parhaimmillaankin tätä on tehotonta. Pahimmassa tapauksessa se on katastrofi, riippuen siitä, mitä muuta metodi tekee.

Katsotaan nyt seuraavaa koodia ja huomataan muutos, joka johtuu yhdistetyn sijoitusoperaattorin käyttämisestä:

```
using System;

class CompoundAssignment2App
{
    protected int[] elements;
    public int[] GetArrayElement()
    {
        return elements;
    }

    CompoundAssignment2App()
    {
        elements = new int[1];
        elements[0] = 42;
    }

    public static void Main()
    {
        CompoundAssignment2App app = new CompoundAssignment2App();

        Console.WriteLine("{0}", app.GetArrayElement()[0]);
        app.GetArrayElement()[0] += 5;
        Console.WriteLine("{0}", app.GetArrayElement()[0]);
    }
}
```

Yhdistetyn sijoitusoperaattorin käyttäminen generoi seuraavan, paljon tehokkaamman MSIL-koodin:

```
// Tehokkaampi x op= y.

.method public hidebysig static void Main() il managed
{
  .entrypoint
  // Code size       76 (0x4c)
  .maxstack  4
  .locals (class CompoundAssignment1App V_0, int32[] V_1)
  IL_0000:  newobj     instance void CompoundAssignment1App::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldstr      "{0}"
  IL_000b:  ldloc.0
  IL_000c:  call instance int32[] CompoundAssignment1App::GetArrayElement()
  IL_0011:  ldc.i4.0
  IL_0012:  ldelema    ['mscorlib']System.Int32
  IL_0017:  box        ['mscorlib']System.Int32
  IL_001c:  call   void ['mscorlib']System.Console::WriteLine
                                    (class System.String, class System.Object)
    IL_0021:  ldloc.0
    IL_0022:  call  instance int32[] CompoundAssignment1App::GetArrayElement()

    IL_0027:  dup
    IL_0028:  stloc.1
    IL_0029:  ldc.i4.0
    IL_002a:  ldloc.1

  IL_002b:  ldc.i4.0
  IL_002c:  ldelem.i4
  IL_002d:  ldc.i4.5
  IL_002e:  add
  IL_002f:  stelem.i4
  IL_0030:  ldstr      "{0}"
  IL_0035:  ldloc.0
  IL_0036:  call     instance int32[] CompoundAssignment1App::GetArrayElement()
  IL_003b:  ldc.i4.0
  IL_003c:  ldelema    ['mscorlib']System.Int32
  IL_0041:  box        ['mscorlib']System.Int32
  IL_0046:  call  void ['mscorlib']System.Console::WriteLine
                                    (class System.String, class System.Object)
  IL_004b:  ret
} // end of method 'CompoundAssignment1App::Main'
```

Näemme, että nyt käytetään MSIL-käskyä *dup*. Käsky tuplaa pinon ylimmän elementin ja tekee siten kopion arvosta, joka on saatu metodin *CompoundAssignment1App.GetArrayElements* paluuarvona.

Tämän tutkimisen tarkoituksena oli havainnollistaa, että vaikka käsitteellisesti $x \mathrel{+}= y$ on sama kuin $x = x + y$, voidaan generoidusta MSIL:stä löytää eroja. Nämä erot tarkoitavat sitä, että sinun tulee miettiä huolellisesti, mitä tapaa käytät eri tilanteissa. Perussääntönä

ja oman mielipiteenäni suosittelen yhdistetyn sijoiutusoperaattorin käyttämistä aina kuin mahdollista.

Lisäysoperaattorit ja vähennysoperaattorit

Jäänteenä siitä lyhennyksestä, joka ensin esiteltiin C-kielessä ja joka sieltä siirtyi sekä C++:aan että Javaan, voit lisäys- ja vähennysoperaattorin avulla lyhyesti ilmaista, että haluat kasvattaa tai vähentää numeerisen muttujan arvoa yhdellä. Siten *i*++ tarkoittaa, että *i*:n nykyiseen arvoon lisätään yksi.

Sekä lisäys- etä vähennysoperaattorista on olemassa kaksi versiota ja se aiheuttaa usein sekaannuksia. Sijoittamalla ++ tai -- merkit joko operaattorin eteen tai jälkeen, määritellään milloin kyseinen toimenpide suoritetaan. Kun käytetään operaattorien etumerkintää, eli ++*a* tai --*a*, suoritetaan lisäys/vähennysoperaatio ensin ja sen jälkeen tehdään muuttujalla mahdollisesti jotain muuta. Kun käytetään takamerkintää, eli *a*++ tai *a*--, tehdään muuttujalla mahdollisesti ensin jotain muuta ja vasta sen jälkeen lisäys/vähennysoperaatio. Katsotaan seuraavaa esierkkiä:

```
using System;

class IncDecApp
{
    public static void Foo(int j)
    {
        Console.WriteLine(“IncDecApp.Foo j = {0}", j);
    }

    public static void Main()
    {
        int i = 1;

        Console.WriteLine(“Before call to Foo(i++) = {0}", i);
        Foo(i++);
        Console.WriteLine(“After call to Foo(i++) = {0}", i);

        Console.WriteLine(“\n”);

        Console.WriteLine(“Before call to Foo(++i) = {0}", i);
        Foo(++i);
        Console.WriteLine(“After call to Foo(++i) = {0}", i);
    }
}
```

Tämä sovellus tuottaa seuraavat tulosrivit:

```
Before call to Foo(i++) = 1
IncDecApp.Foo j = 1
After call to Foo(i++) = 2


Before call to Foo(++i) = 2
IncDecApp.Foo j = 3
After call to Foo(++i) = 3
```

Ero tässä on siis se hetki, milloin operandin arvo muutetaan. Kutsussa *Foo(i++)*, *i*:n arvo välitetään (muuttumattomana) funktiolle *Foo* ja sen *jälkeen*, kun kontrolli palaa *Foo*-metodista, *i*:tä kasvatetaan. Näet tämän seuraavasta MSIL-koodipätkästä. Huomaa, MSIL:n *add*-käskyä ei kutsuta ennen kuin arvo on sijoitettu pinoon.

```
IL_0013:  ldloc.0
IL_0014:  dup
IL_0015:  ldc.i4.1
IL_0016:  add
IL_0017:  stloc.0
IL_0018:  call       void IncDecApp::Foo(int32)
```

Katsotaan nyt etumerkityn lisäysoperaattorin käyttöä kutsussa *Foo(++a)*. Nyt generoitu MSIL näyttää seuraavalta. Huomaa, että MSIL *add*-käskyä kutsutaan *ennen* kuin arvo sijoitetaan pinoon *Foo*-metodin kutsua varten.

```
IL_0049:  ldloc.0
IL_004a:  ldc.i4.1
IL_004b:  add
IL_004c:  dup
IL_004d:  stloc.0
IL_004e:  call       void IncDecApp::Foo(int32)
```

Suhteelliset operaattorit

Useimmat operaattorit palauttavat numeerisen tuloksen. Suhteelliset operaattorit eroavat kuitenkin muista siinä, että ne palauttavat loogisen arvon. Sen sijaan, että ne suorittaisivat joitakin matemaattisia operaatioita joukolla operandeja, ne määrittävät operandien välisen suhteen ja palauttavat arvon *true*, jos suhde on tosi ja arvon *false*, jos suhde ei ole tosi.

Vertailuoperaattorit

Suhteellisten operaattorien joukkoon, josta käytetään nimitystä vertailuoperaattorit, kuuluvat operaattorit: pienempi kuin (<), pienempi tai yhtä suuri kuin (<=), suurempi kuin (>), suurempi tai yhtä suuri kuin (>=), yhtä suuri (==) ja eri suuri (!=). Näiden merkitys on selvä, kun käsitellään numeroita, mutta miten kukin operaattori käsittelee objekteja, ei olekaan enää selvä. Tässä esimerkki:

```
using System;

class NumericTest
{
    public NumericTest(int i)
    {
        this.i = i;
    }

    protected int i;
}

class RelationalOps1App
{
    public static void Main()
    {
        NumericTest test1 = new NumericTest(42);
        NumericTest test2 = new NumericTest(42);

        Console.WriteLine(“{0}", test1 == test2);
    }
}
```

Jos olet Java-ohjelmoija, tiedät mitä tapahtuu. Useimmat C++-ohjelmoijat kuitenkin yllättyvät, kun näkevät esimerkin tulostavan *false*. Muista, että kun instantioit objektin, saat viittauksen keosta varattuun muistialueeseen. Siksi, kun käytät vertailuoperaattoria kahden objektin vertaamiseen, C# ei vertaa objektien sisältöä. Sen sijaan se vertaa näiden kahden objektin osoitetta. Katsotaan vielä MSIL-koodia, jotta ymmärrämme täysin, mitä tässä tapahtuu:

```
.method public hidebysig static void Main() il managed
{
  .entrypoint
  // Code size       39 (0x27)
  .maxstack  3
  .locals (class NumericTest V_0,
           class NumericTest V_1,
           bool V_2)
```

(jatkuu)

```
  IL_0000:  ldc.i4.s   42
  IL_0002:  newobj     instance void NumericTest::.ctor(int32)
  IL_0007:  stloc.0
  IL_0008:  ldc.i4.s   42
  IL_000a:  newobj     instance void NumericTest::.ctor(int32)
  IL_000f:  stloc.1
  IL_0010:  ldstr      “{0}"
  IL_0015:  ldloc.0
  IL_0016:  ldloc.1
  IL_0017:  ceq
  IL_0019:  stloc.2
  IL_001a:  ldloca.s   V_2
  IL_001c:  box        [‘mscorlib’]System.Boolean
  IL_0021:  call   void [‘mscorlib’]System.Console::WriteLine
                                    (class System.String,class System.Object)
  IL_0026:  ret
} // end of method ‘RelationalOps1App::Main’
```

Katso *.locals*-riviä.Kääntäjä määrittelee, että tällä *Main*-metodilla on kolme paikallista muuttujaa. Ensimmäiset kaksi ovat *NumericTest*-objekteja ja kolmas on Boolean-tyyppi. Hyppää nyt riveille *IL_0002* ja *IL_0007*. Täällä MSIL instantioi *test1*-objektin ja *stloc*-käskyllä tallentaa palautetun viittauksen ensimmäiseen paikalliseen muuttujaan. Oleellista tässä on se, että MSIL tallentaa uuden objektin osoitteen. Sitten riveillä *IL_000a* ja *IL_000f* MSIL luo objektin *test2* ja tallentaa palautetun viittauksen toiseen paikalliseen muuttujaan. Lopuksi riveillä *IL_0015* ja *IL_0016* ladataan paikalliset muuttujat pinosta *ldloc*-käskyllä ja rivillä *IL_0017* kutsutaan *ceq*-käskyä, joka vertaa pinon kahta ylimmäistä arvoa (eli viittauksia objekteihin *test1* ja *test2*). Paluuarvo tallennetaan sitten kolmanteen paikalliseen muuttujaan ja tulostetaan myöhemmin kutsulla *System.Console.WriteLine*.

Miten voidaan suorittaa kahden objektin kaikkien jäsenten vertailu? Vastaus löytyy kaikkien .NET Framework -objektien kantaluokasta. *System.Object*-luokalla on metodi nimeltä *Equals* juuri tällaisia tilanteita varten. Esimerkiksi seuraava koodi vertailee kahden objektin sisältöä ja odotusten mukaisesti palauttaa arvon *true*.

```
using System;

class RelationalOps2App
{
    public static void Main()
    {
        Decimal test1 = new Decimal(42);
        Decimal test2 = new Decimal(42);
```

```
        Console.WriteLine("{0}", test1.Equals(test2));
    }
}
```

Huomaa, että RelationalOps1App-esimerkki käytti itse tehtyä luokkaa (*NumericTest*) ja toinen esimerkki käytti .NET-luokkaa (*Decimal*). Syy tähän on se, että *System.Object.Equals*-metodi pitää korvata todellista jäsenkohtaista vertailua varten. Siksi *NumericTest*-luokan *Equals*-metodin käyttäminen ei olisi toiminut, koska emme ole korvanneet sitä. Koska *Decimal*-luokka ei korvaa perittyä *Equals*-metodia, se toimii odotusten mukaan.

Toinen tapa käsitellä objektien vertailua on *operaattorin ylikuormitus* (operator overloading). Operaattorin ylikuormitus määrittelee toiminnon, joka operaattorin "normaalin" toiminnon sijasta määrätyn tyypin objektilla tehdään. Esimerkiksi *String*-objekteilla +-operaattori yhdistää kaksi merkkijonoa eikä tee yhteenlaskua. Tutustumme operaattorin ylikuormitukseen luvussa 13.

Yksinkertaiset sijoitusoperaattorit

Sijoitusoperaattorin vasemmalla puolella olevaa arvoa kutsutaan nimellä *lvalue* ja oikealla puolella olevaa nimellä *rvalue*. *rvalue* voi olla mikä tahansa vakio, muuttuja, numero tai lauseke, joka voidaan ratkaista arvoksi, joka on sama tyyppiä kuin *lvalue*. *lvalue*n tulee olla määritellyn tyypin muuttuja. Syy tähän on se, että arvo kopioidaan oikealta vasemmalle. Siksi pitää olla todellinen, fyysinen tila varattuna muistista, jonne uusi arvo voidaan sijoittaa. Esimerkki voit kirjoittaa *i = 4*, koska *i* esittää fyysistä sijaintia muistissa, joko pinossa tai keossa, riippuen *i*:n todellisesta tyypistä. Et kuitenkaan voi suorittaa käskyä *4 = i*, koska *4* on arvo, ei muuttuja muistissa, jonka sisältöä voidaan muuttaa. Tekninen sääntö C#:ssa sanoo, että *lvalue* voi olla muuttuja, ominaisuus tai indeksoija. Luvusta 7 löydät lisää asiaa ominaisuuksista ja indeksoijista.

Vaikka numeeriset sijoitukset ovat melko selväpiirteisiä, sijoitusoperaatiot, jotka liittyvät objekteihin, ovat paljon monimutkaisempia. Muista, että kun käsittelet objekteja, et käsittele yksinkertaisia pinossa olevia arvoja, joita on helppo kopioida ja siirtää eri puolille. Kun käsitellään objekteja, sinulla on vain viittaus keosta varattuun muistiin. Siksi, kun yrität sijoittaa objektia (tai mitä tahansa viittaustyyppiä) muuttujaan, et kopioi tietoja kuten teet arvotyypeillä. Kopioit yksinkertaisesti viittauksen paikasta toiseen.

Sanotaan, että sinulla on kaksi objektia: *test1* ja *test2*. Jos kirjoitat *test1 = test2*, *test1* ei ole kopio *test2*:sta. Se on sama asia! *test1*-objekti osoittaa samaan muistialueeseen kuin *test2*. Siksi kaikki *test1*-objektiin tehtävät muutokset muuttavat myös *test2*-objektia. Tässä ohjelma, joka osoittaa tämän:

```
using System;

class Foo
{
    public int i;
}

class RefTest1App
{
    public static void Main()
    {
        Foo test1 = new Foo();
        test1.i = 1;

        Foo test2 = new Foo();
        test2.i = 2;

        Console.WriteLine(“BEFORE OBJECT ASSIGNMENT”);
        Console.WriteLine(“test1.i={0}", test1.i);
        Console.WriteLine(“test2.i={0}", test2.i);
        Console.WriteLine(“\n”);

        test1 = test2;

        Console.WriteLine(“AFTER OBJECT ASSIGNMENT”);
        Console.WriteLine(“test1.i={0}", test1.i);
        Console.WriteLine(“test2.i={0}", test2.i);
        Console.WriteLine(“\n”);

        test1.i = 42;

        Console.WriteLine(“AFTER CHANGE TO ONLY TEST1 MEMBER”);
        Console.WriteLine(“test1.i={0}", test1.i);
        Console.WriteLine(“test2.i={0}", test2.i);
        Console.WriteLine(“\n”);
    }
}
```

Suorita tämä koodi ja saat seuraavanlaisen tuloksen:

```
BEFORE OBJECT ASSIGNMENT
test1.i=1
test2.i=2
```

```
AFTER OBJECT ASSIGNMENT
test1.i=2
test2.i=2

AFTER CHANGE TO ONLY TEST1 MEMBER
test1.i=42
test2.i=42
```

Käydään läpi tämä esimerkki ja katsotaan, mitä tapahtuu missäkin vaiheessa. *Foo* on yksinkertainen luokka, joka sisältää yhden jäsenen nimeltä *i*. *Main*-metodissa luodaan tästä luokasta kaksi insanssia, *test1* ja *test2*. Objektin luonnin jälkeen asetetaan jäsenen *i* arvo, ensin arvoon 1 ja toisessa objetissa arvoon 2. Tässä vaiheessa tulostamme arvot ja ne ovat odotusten mukaisesti 1 ja 2. Nyt alkaa hauskuus. Seuraavalla rivillä sijoitetaan *test2* objektiin *test1*. Java-ohjelmoijat tietävät, mitä tapahtui. Useimmat C++-ohjelmoijat kuitenkin olettavat, että *test1*-objektin jäsenen *i* arvo on nyt sama kuin *test2*-objektin jäsenen *i* arvo (olettaen, että koska sovellus kääntyi, on täytynyt tapahtua jonkinlainen jäsenkohtainen kopiointitoimenpide). Itse asiassa siltä näyttää, kun tulostetaan molempien objektien jäsenen arvot. Näiden objektien välinen suhde menee kuitenkin tätäkin syvemmälle. Koodi sijoittaa seuraavaksi arvon 42 *test1.i*:hin ja tulostaa taas molempien objektien jäsenten arvot. Mitä?! *test1*-objektin muuttaminen muutti myös *test2*-objektia. Tämä johtuu siitä, että aiemmin nimellä *test1* tunnettua objektia ei ole enää olemassa. Sijoituksessa *test1 = test2* menetettiin objekti *test1* koska siihen ei enää viitata ohjelmassa ja roskienkeruu (GC) on käynyt siivoamassa sen. *test1* ja *test2* osoittavat nyt samaan muistialueeseen keossa. Siksi jompaan kumpaan tehty muutos näkyy myös toisen muuttujan muutoksena.

Huomaa tulosteen kahdelta viimeiseltä riviltä, että vaikka koodi asettaa vain *test1.i*:n arvon, myös *test2.i*:n arvo muuttuu. Tämä johtuu siis siitä, että molemmat muuttujat osoittavat nyt samaan paikkaan muistissa, aivan kuten Java-ohjelmoijat osasivat odottaakin. Se on kuitenkin aivan eri, mitä C++-ohjelmoija odottaa, koska C++:ssa objektien kopiointitoiminto tarkoittaa juuri sitä, jonka jälkeen kullakin muuttujalla on oma yksilöllinen kopionsa jäsenistä, jolloin yhden objektin muuttamisella ei ole vaikutusta toiseen. Koska tämä on olellinen asia ymmärtää, kun työskentelee objekteilla C#:ssa, käydään vielä nopeasti läpi, mitä tapahtuu, kun välitetään objekti metodille:

```
using System;

class Foo
{
```

(jatkuu)

```
    public int i;
}

class RefTest2App
{
    public void ChangeValue(Foo f)
    {
        f.i = 42;
    }

    public static void Main()
    {
        RefTest2App app = new RefTest2App();

        Foo test = new Foo();
        test.i = 6;

        Console.WriteLine(“BEFORE METHOD CALL”);
        Console.WriteLine(“test.i={0}", test.i);
        Console.WriteLine(“\n”);

        app.ChangeValue(test);

        Console.WriteLine(“AFTER METHOD CALL”);
        Console.WriteLine(“test.i={0}", test.i);
        Console.WriteLine(“\n”);
    }
}
```

Useimmissa kielissä, Javaa lukuunottamatta, tämä koodi aiheuttaisia *test*-objektin kopion luomisen *RefTest2App.ChangeValue*-metodin paikalliseen pinoon. Jos niin tapahtuisi, *Main*-metodissa luotu *test*-objekti ei "näkisi" *ChangeValue*-metodin *f*-objektiin tekemiä muutoksia. Tässä tapahtui niin, että *Main*-metodi välitti viittauksen keossa sijaitsevaan *test*-objektiin. Kun *ChangeValue*-metodi käsittelee paikallista *f.i*-muuttujaansa, se käsittelee samalla suoraan *Main*-metodin *test*-objektia.

Yhteenveto

Oleellinen osa jokaista ohjelmointikieltä on se, miten se käsittelee sijoituksia sekä matemaattisia, suhteellisia ja loogisia operaatioita suorittaessaan jokaisessa todellisessa sovelluksessa tarvittavia perustehtäviä. Näitä toimintoja ohjataan koodissa operaattoreilla. Operaattorien vaikutuksen koodissa määrittelevät suoritusjärjestys sekä oikea ja vasen liitettävyys. Sen lisäksi, että tarjoaa tehokkaan joukon valmiita operaattoreita, C# mahdollistaa niiden laajentamisen käyttäjän määrittelemien toteutusten kautta. Tämä on asia, josta puhutaan luvussa 13.