13

Operaattorin ylikuormitus ja käyttäjän muunnokset

Luvussa 7, "Ominaisuudet, taulukot ja indeksoijat," opit, miten luokan yhteydessä käytetään ohjelmallisesti []-operaattoria, jolloin objektia voidaan käsitellä kuin se olisi taulukko. Tässä luvussa tutkimme siihen läheisesti liittyviä C#:n ominaisuuksia, joiden avulla voit tehdä rakenteita ja luokkarajapintoja, joita on helpompi ja luonnollisempi käyttää: operaattorin ylikuormitusta ja käyttäjän omia muunnoksia. Aloitan johdannolla operaattorin ylikuormitukseen ja sen tarjoamiin etuihin ja sen jälkeen katsomme syntaksia, jolla operaattorin oletuskäyttäytyminen määritellään uudelleen sekä todellisemman esimerkin, jossa ylikuormitan +-operaattorin yhdistämään useita *Invoice*-objekteja. Sitten näytän luettelon niistä yksioperandisista ja binaarisista operaattoreista, jotka voidaan ylikuormittaa sekä kerron muutamasta asiaan liittyvästä rajoituksesta. Operaattorin ylikuormituksen käsittely lopetetaan muutamiin suunnitteluohjeisiin, jotka tulee huomioida, kun luokan suunnittelun yhteydessä päätetään mahdollisista operaattorien ylikuormituksista. Toisena asiana opit uuden käsitteen, käyttäjän muunnos (user-defined conversion). Aloitan taas johdannolla kertomalla ominaisuuden perusteet ja sitten sukellan luokkaan, joka näyttää, miten voit muunnoksen avulla tehdä tyyppimuunnoksen *struct*- tai *class*-tyypistä toiseen *struct*- tai *class*-tyyppiin tai C#:n perustyyppiin.

Operaattorin ylikuormitus

Operaattorin ylikuormitus mahdollistaa olemassa olevan C#-operaattorin uudelleenmäärittämisen käytettäväksi käyttäjän omien tyyppien kanssa. Operaattorin ylikuormitusta sanotaan joskus pelkäksi sokerikuorrutukseksi, koska se itse asiassa on vain erilainen tapa kutsua metodia. On myös sanottu, että se ei pohjimmiltaan tuo kieleen mitään uutta. Vaikka se onkin teknisessä mielessä totta, operaattorin ylikuormitus auttaa olioperusteisen ohjelmoinnin tärkeimmässä perusteessa: abstraktiossa.

Oletetaan, että sinun pitää yhdistää kokoelma määrätyn asiakkaan laskuja. Operaattorin ylikuormitusta käyttämällä voit kirjoittaa seuraavanlaisen koodin, jossa +=-operaattori on ylikuormitettu.

```
Invoice summaryInvoice = new Invoice();
foreach (Invoice invoice in customer.GetInvoices())
{
    summaryInvoice += invoice;
}
```

Tällaisen koodin etu on sen luonnollisuudessa ja siinä tosiseikassa, että käyttäjä on abstraktoitu siitä, miten laskujen yhdistäminen yksityiskohtaisesti tapahtuu. Yksinkertaisesti sanoen, operaattorin ylikuormitus auttaa kirjoittamaan ohjelmia, jotka ovat edullisempia tehdä ja ylläpitää.

Syntaksi ja esimerkki

Kuten sanoin, operaattorin ylikuormitus tarkoittaa metodin kutsumista. Määritelläksesi operaattorin luokassa uudelleen, sinun tulee vain käyttää seuraavaa mallia, missä *op* on ylikuormitettava operaattori:

public static *retval* operator*op* (*object1* [, *object2*])

Pidä mielessä seuraavat tosiasiat, kun käytät operaattorin ylikuormitusta:

- Kaikki ylikuormitettavat operaattorit tulee määritellä määreellä *public* tai *static*.
- Teknisesti *retval* (paluuarvo) voi olla mitä tahansa tyyppiä. On kuitenkin yleinen käytäntö, että paluuarvo on sama tyyppi, missä metodi on määritelty sillä poikkeuksella, että *true* ja *false*-operaattorin tulee aina palauttaa Boolean-tyyppi.
- Välitettävien parametrien (*object1*, *object2*) määrä riippuu ylikuormitettavan operaattorin tyypistä. Jos ylikuormitetaan yksioperandista operaattoria, sillä on yksi parametri. Jos operaattori on binaarinen (eli sillä on kaksi operandia), välittään kaksi parametria.
- Yksioperandisen operaattorin tapauksessa metodin parametrin tulee olla saman tyyppinen kuin luokka tai tietue, jossa metodi määritellään. Toisin sanoen, jos uudelleenmäärittelet !-operaattorin luokassa nimeltä *Foo*, pitää metodin ainoan parametrin tyyppi olla *Foo*.
- Jos ylikuormitettava operaattori on binaarioperaattori, sen ensimmäisen parametrin tulee olla samaa tyyppiä, jossa metodi määritellään. Toinen parametri voi olla mitä tahansa tyyppiä.

Edellisen kappaleen pseudokoodissa käytin +=-operaattoria *Invoice*-luokassa. Syystä, jonka kohta ymmärrät, et voi tosiasiallisesti ylikuormittaa näitä yhdistelmäoperaattoreita. Voit ylikuormittaa vain "kantaoperaattorin", tässä tapauksessa siis operaattorin +. Tässä rakenne, jolla määritellään *Invoice*-luokan *operator*+-metodi:

```
public static Invoice operator+ (Invoice invoice1, Invoice invoice2)
{
    // Luo uusi Invoice-objekti.
    // Lisää haluttu sisältö
    // invoice1-objektista uuteen Invoice-objektiin.
    // Lisää haluttu sisältö
    // invoice2-objektista uuteen Invoice-objektiin.
    // Palauta uusi Invoice-objekti.
 }
```

Katsotaan nyt vähän sisällökkäämpää esimerkkiä. Esimerkissä on kaksi pääluokkaa: *Invoice* ja *InvoiceDetailLine*. *Invoice*-luokassa on *ArrayList*-tyyppinen jäsenmuuttuja, joka esittää kaikkien laskun rivien kokoelman. Jotta kahden laskun (invoice) rivien yhdistäminen olisi mahdollista, ylikuormitin +-operaattorin. (Katso yksityiskohtia alla olevasta *operator*+-metodista). *Invoice.operator+*-metodi luo uuden *Invoice*-objektin ja käy läpi kummankin laskun laskurivien taulukon lisäten kunkin rivin uuteen *Invoice*-objektiin. Tämä uusi *Invoice*-objekti palautetaan sitten kutsujalle. Todellisessa laskutusmoduulissa tämä oli luonnollisesti paljon monimutkaisempi toimenpide, mutta esimerkki kuitenkin näyttää suhteellisen todentuntuisesti, miten operaattorin ylikuormitusta voi käyttää.

```
using System;
using System.Collections;

class InvoiceDetailLine
{
    double lineTotal;
    public double LineTotal
    {
        get
        {
            return this.lineTotal;
        }
    }

    public InvoiceDetailLine(double LineTotal)
    {
        this.lineTotal = LineTotal;
```

(jatkuu)

```
    }
}

class Invoice
{
    public ArrayList DetailLines;

    public Invoice()
    {
        DetailLines = new ArrayList();
    }

    public void PrintInvoice()
    {
        Console.WriteLine("\nLine Nbr\tTotal");

        int i = 1;
        double total = 0;
        foreach(InvoiceDetailLine detailLine in DetailLines)
        {
            Console.WriteLine("{0}\t\t{1}", i++, detailLine.LineTotal);
            total += detailLine.LineTotal;
        }

        Console.WriteLine("=====\t\t===");
        Console.WriteLine("Total\t\t{1}", i++, total);
    }

    public static Invoice operator+ (Invoice invoice1, Invoice invoice2)
    {
        Invoice returnInvoice = new Invoice();

        foreach (InvoiceDetailLine detailLine in invoice1.DetailLines)
        {
            returnInvoice.DetailLines.Add(detailLine);
        }

        foreach (InvoiceDetailLine detailLine in invoice2.DetailLines)
        {
            returnInvoice.DetailLines.Add(detailLine);
        }
        return returnInvoice;
    }
}

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

```
    {
        Invoice i1 = new Invoice();
        for (int i = 0; i < 2; i++)
        {
            i1.DetailLines.Add(new InvoiceDetailLine(i + 1));
        }

        Invoice i2 = new Invoice();
        for (int i = 0; i < 2; i++)
        {
            i2.DetailLines.Add(new InvoiceDetailLine(i + 1));
        }

        Invoice summaryInvoice = i1 + i2;

        summaryInvoice.PrintInvoice();
    }
}
```

Ylikuormitettavat operaattorit

Vain seuraavat yksioperandiset ja binaariset operaattorit voidaan ylikuormittaa.

Yksioperandiset: +, -, !, ~, ++, --, *true*, *false*

Binaarioperandit: +, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, <=

> Huomaa Pilkkua käytetään tässä erottamaan eri ylikuormitettavat operandit toisistaan. Pilkku-operaattoria, jota käytetään *for*-käskyssä ja metodikutsuissa, ei voi ylikuormittaa.

Rajoitukset operaattorin ylikuormituksessa

Ei ole mahdollista ylikuormittaa sijoitusoperaattoria =. Kun ylikuormitat binaarioperaattoria, sen yhdistelmäsijoitusoperaattori ylikuormitetaan kuitenkin epäsuorasti. Jos esimerkiksi ylikuormitat +-operaattorin, +=-operaattori kuormitetaan epäsuorasti, kun käyttäjän määrittämää *operator*+-metodia kutsutaan.

[]-operaattoreita ei voi ylikuormittaa. Kuten kuitenkin näit luvussa 7, käyttäjän luomat objektit voidaan indeksoida indeksoijien avlla.

Tyyppimuunnoksessa käytettävät sulut eivät ole ylikuormitettavissa. Sen sijaan sinun tulee käyttää muunnos-operaattoreita, joita myös käyttäjän muunnoksiksi sanotaan. Asiasta puhutaan tämän luvun loppuosassa.

Operaattoreita, joita ei C#:ssa ole määritelty, ei voida ylikuormittaa. Et voi esimerkiksi määritellä merkkejä ** tarkoittamaan potenssiin korotusta, koska C# ei määrittele **-operaattoria. Myöskään operaattorin syntaksia ei voi muokata. Et voi muuttaa binaarista *-operaattoria ottamaan kolmea parametria, koska sen oletussyntaksi tarvitsee vain kaksi paramatria. Lopuksi, operaattorien suoritusjärjestystä ei voi muuttaa. Suoritusjärjestyssäännöt ovat pysyviä, katso luvusta 10, "Lausekkeet ja operaattorit," lisää näistä säännöistä.

Suunnitteluohjeita

Olet nähnyt, mikä operaattorin ylikuormitus on ja miten sitä käytetään C#:ssa, joten opiskellaan tämän käyttökelpoisen ominaisuuden usein unohtuvaa asiaa: suunnitteluohjeita. Sinun on yritettävä pysyä erossa luonnollisesta taipumuksesta käyttää uutta ominaisuutta sen vuoksi, että se on olemassa. Tätä ilmiötä sanotaan joskus "ongelman etsimiseksi ratkaisuun." On aina hyvää suunnittelua muistaa sanonta "koodia luetaan useammin kuin kirjoitetaan." Pidä luokan käyttäjät mielessä, kun päätät, ylikuormitatko operaattorin ja koska sen teet. Tässä nyrkkisääntö: sinun tulee ylikuormittaa operaattori vain, jos se tekee luokan rajapinnan luonnollisemmaksi käyttää. Se esimerkiksi sopii täydellisesti tilanteeseen, jossa laskuja pitää yhdistää.

Älä myöskään unohda, että sinun tulee ajatella kuin luokkasi käyttäjä. Sanotaan, että teet *Invoice*-luokkaa ja haluat, että käyttäjä voi antaa laskulle alennuksen. Sinä ja minä tiedämme, että laskuun lisätään hyvitysrivi, mutta kapseloinnin idea on siinä, että luokkasi käyttäjän ei tarvitse tietää luokan toteutuksen täsmällisiä yksityiskohtia. Siksi *-operaattorin ylikuormittaminen seuraavalla tavalla voi olla hyvä idea, koska se palvelee *Invoice*-luokan rajapintaa luonnollisella ja käyttökelpoisella tavalla:

```
invoice *= .95; // 5% alennus.
```

Käyttäjän muunnokset

Mainitsin aiemmin, että tyyppimuunnokseen käytettäviä sulkuja ei voi ylikuormittaa ja että sen sijaan tulee käyttää muunnoksia. Lyhyesti sanottuna, käyttäjän muunnosten avulla voit määritellä tietueille tai luokille muunnoksia, joissa *struct* tai *class* muunnetaan toiseksi tietueeksi, luokaksi tai C#:n perustyypiksi. Miksi ja milloin tällaiseen on tarvetta? Sanotaan, että sinun pitää käyttää sovelluksessasi Celsius- ja Fahrenheit-lämpötila-asteikkoja siten,

että niiden muuntaminen toiseksi onnistuu helposti. Tekemällä käyttäjän muunnoksen voit käyttää seuraavaa syntaksia:

```
Fahrenheit f = 98.6F;
Celsius c = (Celsius)f; // Käyttäjän muunnos.
```

Tässä ei ole mitään toiminnallista etua seuraavaan syntaksiin nähden, mutta se on luonnollisempi kirjoittaa ja helpompi lukea.

```
Fahrenheit f = 98.6F;
Celsius c = f.ConvertToCelsius();
```

Syntaksi ja esimerkki

Käyttäjän muunnoksen syntaksi käyttää *operator*-avainsanaa muunnoksen määrittelyssä seuraavasti:

public static implicit operator *conv-type-out* (*conv-type-in operand*)

public static explicit operator *conv-type-out* (*conv-type-in operand*)

Muunnoksen syntaksiin liittyy vain muutama sääntö:

- Jokaisen muunnosmetodin määreen tulee olla *static*. Muunnoksia voi yhdessä luokassa tai tietueessa olla vaikka kuinka monta.
- Muunnos pitää määritellä joko sanalla *implicit* tai *explicit*. *implicit*-avainsana tarkoittaa, että muunnos tehdään automaattisesti eikä käyttäjän tarvitse sitä erikseen määrätä. Vastaavasti *explicit*-avainsana ilmoittaa, että käyttäjän tulee tehdä muunnos explisiittisesti.
- Kaikkien muunnosten tulee ottaa parametrinaan tyyppi, jossa muunnos määritellään tai palauttaa tyyppi, jossa muunnos määritellään.
- Kuten operaattorin ylikuormituksessa, käytetään *operator*-avainsanaa metodin otsikossa mutta ilman lisäoperaattoria.

Kun ensimmäisen kerran luin nämä säännöt, minulla ei ollut harmainta aavistustakaan mitä pitää tehdä, joten katsotaan esimerkkiä. Siinä meillä on kaksi tietuetta (*Celsius* ja *Fahrenheit*), joiden avulla luokan käyttäjä voi muuntaa *float*-tyyppisen arvon jompaan kumpaan lämpötila-asteikkoon. Esitän ensin Celsius-muunnoksen ja kerron siitä muutamia seikkoja ja sen jälkeen näet täydellisen toimivan sovelluksen.

```
struct Celsius
{
```

```
    public Celsius(float temp)
    {
        this.temp = temp;
    }

    public static implicit operator Celsius(float temp)
    {
        Celsius c;
        c = new Celsius(temp);
        return(c);
    }

    public static implicit operator float(Celsius c)
    {
        return((((c.temp - 32) / 9) * 5));
    }

    public float temp;
}
```

Ensimmäinen päätös, jonka näet, on tietueen käyttäminen luokan sijasta. Minulla ei ole siihen muuta todellista syytä kuin se, että luokan käyttäminen on muistin käytön kannalta tehottomampaa ja toisaalta luokkaa ei tässä välttämättä tarvita, koska *Celsius*-tietue ei tarvitse mitään C#:n luokkakohtaisia ominaisuuksia, kuten esimerkiksi periytymistä.

Huomioi seuraavaksi, että määrittelin muodostimen, joka ottaa ainoaksi parametrikseen *float*-tyypin muuttujan. Tämä arvo tallennetaan *temp*-nimiseen jäsenmuuttujaan. Katso nyt muunnosoperaattoria, jonka määrittely on heti tietueen muodostimen jäljessä. Tämä on metodi, jota kutsutaan, kun asiakas tekee tyyppimuunnoksen *float*-tyypistä *Celsius*-tyypiksi, tai käyttää *float*-tyyppiä sellaisessa paikassa, esimerkiksi metodikutsussa, jossa pitäisi käyttää *Celsius*-tietuetta. Tämän metodin ei tarvitse tehdä paljon ja se onkin pelkkä kaavan sisältävä koodi, jota voidaan käyttää useissa perusmuunnoksissa. Yksinkertaisesti instantioin *Celsius*-tietueen ja sen jälkeen palautan sen. *return*-kutsu on se, joka aiheuttaa tietueen viimeisen metodin kutsumisen. Kuten näet, metodi yksinkertaisesti tekee matemaattisen kaavan mukaisen laskutoimituksen muuntaen Fahrenheit-arvon Celcius-arvoksi.

Tässä koko sovellus, jossa on mukana myös *Fahrenheit*-tietue:

```
using System;

struct Celsius
{
    public Celsius(float temp)
    {
        this.temp = temp;
    }
```

```
    public static implicit operator Celsius(float temp)
    {
        Celsius c;
        c = new Celsius(temp);
        return(c);
    }

    public static implicit operator float(Celsius c)
    {
        return((((c.temp - 32) / 9) * 5));
    }

    public float temp;
}

struct Fahrenheit
{
    public Fahrenheit(float temp)
    {
        this.temp = temp;
    }

    public static implicit operator Fahrenheit(float temp)
    {
        Fahrenheit f;
        f = new Fahrenheit(temp);
        return(f);
    }

    public static implicit operator float(Fahrenheit f)
    {
        return((((f.temp * 9) / 5) + 32));
    }

    public float temp;
}

class Temp1App
{
    public static void Main()
    {
        float t;
```

(jatkuu)

```
        t=98.6F;
        Console.Write("Conversion of {0} to Celsius = ", t);
        Console.WriteLine((Celsius)t);

        t=0F;
        Console.Write("Conversion of {0} to Fahrenheit = ", t);
        Console.WriteLine((Fahrenheit)t);
    }
}
```

Jos käännät ja suoritat tämän sovelluksen, saat seuraavat tulokset:

```
Conversion of 98.6 to Celsius = 37
Conversion of 0 to Fahrenheit = 32
```

Tämä toimii hyvin ja mahdollisuus toteuttaa muunnos kirjoittamalla *(Celsius)98.6F* on ilman muuta fiksumpi tapa kuin jonkin staattisen metodin kutsuminen. Mutta huomaa, että voit välittää parametrina näille muunnoksille vain *float*-tyyppisen arvon. Yllä oleva sovelluksessa seuraava ei käänny:

```
Celsius c = new Celsius(55);
Console.WriteLine((Fahrenheit)c);
```

Koska ei myöskään ole Celsius-muunnosmetodia, joka ottaisi parametrinaan *Fahrenheit*-tietueen (ja päinvastoin), koodin pitää olettaa, että sille välitetty arvo on muunnettava arvo. Toisin sanoen, jos kutsun *(Celsius)98.6F*, saan tuloksen 37. Jos tuo arvo sitten välitetään takaisin muunnosmetodille, sillä ei ole mitään tapaa tietää, että arvo on jo muunnettu ja on loogisesti Celsius-asteina, muunnosmetodin kannalta se on vain *float*-tyyppinen arvo. Ja siksi se muunnetaan uudelleen. Tämän vuoksi meidän pitää muokata sovellusta niin, että kukin tietue voi ottaa kelvollisena parametrinaan toisen tietueen.

Kun alun perin ajattelin tämän tekemistä, kavahdin ajatusta, koska mietin miten vaikeaa sellainen olisi tehdä. Osoittautui kuitenkin, että se äärimmäisen helppoa. Tässä muutettu koodi jälkikommentein varustettuna:

```
using System;

class Temperature
{
    public Temperature(float Temp)
    {
        this.temp = Temp;
    }

    protected float temp;
    public float Temp
```

```
    {
        get
        {
            return this.temp;
        }
    }
}

class Celsius : Temperature
{
    public Celsius(float Temp)
        : base(Temp) {}

    public static implicit operator Celsius(float Temp)
    {
        return new Celsius(Temp);
    }

    public static implicit operator Celsius(Fahrenheit F)
    {
        return new Celsius(F.Temp);
    }

    public static implicit operator float(Celsius C)
    {
        return((((C.temp - 32) / 9) * 5));
    }
}

class Fahrenheit : Temperature
{
    public Fahrenheit(float Temp)
        : base(Temp) {}

    public static implicit operator Fahrenheit(float Temp)
    {
        return new Fahrenheit(Temp);
    }

    public static implicit operator Fahrenheit(Celsius C)
    {
        return new Fahrenheit(C.Temp);
    }

    public static implicit operator float(Fahrenheit F)
    {
        return((((F.temp * 9) / 5) + 32));
    }
```

(jatkuu)

```
}

class Temp2App
{
    public static void DisplayTemp(Celsius Temp)
    {
        Console.Write("Conversion of {0} {1} to Fahrenheit = ",
            Temp.ToString(), Temp.Temp);
        Console.WriteLine((Fahrenheit)Temp);
    }

    public static void DisplayTemp(Fahrenheit Temp)
    {
        Console.Write("Conversion of {0} {1} to Celsius = ",
            Temp.ToString(), Temp.Temp);
        Console.WriteLine((Celsius)Temp);
    }

    public static void Main()
    {
        Fahrenheit f = new Fahrenheit(98.6F);
        DisplayTemp(f);

        Celsius c = new Celsius(0F);
        DisplayTemp(c);
    }
}
```

Ensimmäinen tekemäni muutos oli se, että muutin *Celsius* ja *Fahrenheit* -tyypit tietueesta luokaksi. Tein sen, jotta minulla olisi kaksi esimerkkiä, toinen tietueilla toteutettu ja toinen luokilla. Mutta käytännöllisempi syy oli jakaa *temp*-jäsenmuuttuja periyttämällä *Celsius* ja *Fahrenheit* -luokat samasta *Temperature*-kantaluokasta. Nyt voin myös käyttää *System.Object*-luokasta perittyä *ToString*-metodia sovelluksen tulostuksessa.

Ainoa muu huomioitava muutos oli sellaisen muunnoksen lisääminen kuhunkin lämpötila-asteikkoon, joka ottaa parametrinaan toisen asteikon tyypin. Huomaa, miten samanlaista koodi kahden Celsius-muunnosmetodin välillä on:

```
public static implicit operator Celsius(float temp)
{
    Celsius c;
    c = new Celsius(temp);
    return(c);
}
public static implicit operator Celsius(Fahrenheit f)
{
    Celsius c;
```

```
    c = new Celsius(f.temp);
    return(c);
}
```

Ainoa tehtävä, joka minun piti tehdä eri tavalla, oli muuttaa välitettävä parametri ja hakea lämpötila välitetyltä objektilta kovakoodatun *float*-tyypin sijasta. Tämä on syy, miksi aiemmin huomautin, kuinka helppoa muunnosmetodien tekeminen sitten on, kun tuntee perusasiat.

Yhteenveto

Operaattorin ylikuormitus ja käyttäjän muunnokset ovat hyödyllisiä, kun tehdään hyviä rajapintoja luokalle. Kun käytät operaattorin ylikuormitusta, pidä mielessä asiaan liittyvät rajoitukset. Esimerkiksi se, että vaikka et voikaan ylikuormittaa =-operaattoria, niin kun binaarioperaattori ylikuormitetaan, sen yhdistelmäsijoitus ylikuormitetaan automaattisesti. Seuraa suunnitteluohjeita, kun päätät kunkin ominaisuuden käyttämisestä. Pidä mielessä luokan käyttäjä, kun mietit, ylikuormitatko operaattorin tai operaattoreita vai et. Tuntemalla käyttäjän ajatuksia ja tarpeita, voit käyttää näitä tehokaista ominaisuuksia ja tehdä luokan, jonka määrättyjä toimintoja voidaan suorittaa luonnollisella syntaksilla.