6

Metodit

Kuten opit luvussa 1, ”Olioperusteisen ohjelmoinnin teoria,” luokka on kapseloitu joukko tietoja ja niitä käsitteleviä metodeja. Toisin sanoen metodit antavat luokalle sen toiminnalliset ominaisuudet ja koodaamme ne niiden toimintojen mukaan, joita haluamme luokan tekevän puolestamme. Tähän asti olen jättänyt kaikki C#-metodien määrittelyyn ja kutsumiseen liittyvät tarkemmat yksityiskohdat kertomatta. Ne tehdään nyt tässä luvussa, käyn läpi avainsanat *ref* ja *out* metodin parametreissa ja miten niiden avulla voit määritellä metodin palauttamaan kutsujalle enemmän kuin yhden arvon. Opit myös miten metodi ylikuormitetaan, jotta saman nimiset metodit voivat toimia eri tavalla riippuen niille välitettävien parametrien tyypistä ja/tai määrästä. Sen jälkeen opit käsittelemään tilanteita, joissa et tiedä todellista parametrien määrään ennen kuin suorituksen aikana. Päätän luvun puhumalla virtuaalisista metodeista, jotka perustuvat luvussa 5, ”Luokat, ” käsiteltyyn periytymiseen, ja miten määrittelet staattiset metodit.

ref- ja *out*-tyyppiset parametrit

Kun C#:ssa yritetään hakea tietoja metodin avulla, voit saada vain paluuarvon. Siksi ensisilmäyksellä näyttää, että voit saada takaisin yhdellä metodikutsulla vain yhden arvon. Yhden metodikutsun tekeminen aina yhtä arvoa varten on monissa tilanteissa ilmiselvästi turhauttavaa. Sanotaan esimerkiksi, että sinulla on Color-luokka, joka esittää annettua väriä ja sen kolmea RGB-standardin mukaista arvoa. Paluuarvoa käyttämällä sinun pitäisi kirjoittaa kolme seuraavalla sivulla olevan metodin tapaista metodia, jotta saisit haettua kaikki kolme arvoa.

```
// Olettaen, että color on Color-luokan instanssi.
int red = color.GetRed();
int green = color.GetGreen();
int blue = color.GetBlue();
```

Mutta haluamme jotain tällaista:

```
int red;
int green;
int blue;
color.GetRGB(red, green, blue);
```

Tässä on kuitenkin ongelma. Kun kutsutaan *color.GetRGB*-metodia, *red*, *green* ja *blue* -parametrien arvot kopioidaan metodin paikalliseen pinoon eikä metodin mahdollisesti tekemiä muutoksia tehdä kutsuvan ohjelman muuttujiin.

C++:ssa tämä ongelma kierretään niin, että kutsuva metodi välittää osoittimen tai viittauksen muuttujiin, jolloin metodi käsittelee kutsujan tietoja. Ratkaisu on C#:ssa samanlainen. C# tarjoaa todellisuudessa kaksi samanlaista ratkaisua. Ensimmäinen käyttää avainsanaa *ref*. Tämä avainsana kertoo C#-kääntäjälle, että välitettävät parametrit osoittavat samaan muistialueeseen kuin muuttujat kutsuvassa ohjelmassa. Tällöin, jos metodi muuttaa näiden muuttujien arvoa, on kutsuvan koodin muuttujien arvo muutettu, kun ohjelma palaa metodista. Seuraava koodi esittää *ref*-avainsanan käyttöä *Color*-luokan esimerkissä:

```
using System;

class Color
{
    public Color()
    {
        this.red = 255;
        this.green = 0;
        this.blue = 125;
    }

    protected int red;
    protected int green;
    protected int blue;

    public void GetColors(ref int red, ref int green, ref int blue)
    {
        red = this.red;
        green = this.green;
        blue = this.blue;
    }
}
```

```
class RefTest1App
{
    public static void Main()
    {
        Color color = new Color();
        int red;
        int green;
        int blue;
        color.GetColors(ref red, ref green, ref blue);
        Console.WriteLine("red = {0}, green = {1}, blue = {2}",
red, green, blue);
    }
}
```

ref-avainsanan käytössä on yksi haittapuoli ja itse asiassa sen rajoituksesta johtuen ei yllä oleva koodi edes käänny. Kun käytät *ref*-avainsanaa, sinun pitää alustaa välitettävät parametrit ennen metodin kutsumista. Sen vuoksi koodi pitää muuttaa tällaiseksi, jotta se toimisi:

```
using System;

class Color
{
    public Color()
    {
        this.red = 255;
        this.green = 0;
        this.blue = 125;
    }

    protected int red;
    protected int green;
    protected int blue;

    public void GetColors(ref int red, ref int green, ref int blue)
    {
        red = this.red;
        green = this.green;
        blue = this.blue;
    }
}

class RefTest2App
{
    public static void Main()
```

(jatkuu)

```
    {
        Color color = new Color();
        int red = 0;
        int green = 0;
        int blue = 0;
        color.GetColors(ref red, ref green, ref blue);
        Console.WriteLine("red = {0}, green = {1}, blue = {2}",
red, green, blue);
    }
}
```

Tässä esimerkissä niiden muuttujien alustaminen, jotka aiotaan ylikirjoittaa, näyttää turhalta, eikö näytäkin? Siksi C# tarjoaa vaihtoehtoisen tavan välittää parametreja, joiden arvoa tulee voida metodissa muuttaa: *out*-avainsanan. Tässä sama *Color*-luokan esimerkki käyttäen *out*-avainsanaa:

```
using System;

class Color
{
    public Color()
    {
        this.red = 255;
        this.green = 0;
        this.blue = 125;
    }

    protected int red;
    protected int green;
    protected int blue;

    public void GetColors(out int red, out int green, out int blue)
    {
        red = this.red;
        green = this.green;
        blue = this.blue;
    }
}

class OutTest1App
{
    public static void Main()
    {
        Color color = new Color();
        int red;
        int green;
        int blue;
```

```
        color.GetColors(out red, out green, out blue);
        Console.WriteLine("red = {0}, green = {1}, blue = {2}",
                    red, green, blue);
    }
}
```

Ainoa ero *ref* ja *out*-avainsanojen välillä on se, että *out*-avainsana ei vaadi välitettävän muuttujan alustamista ennen kutsua. Milloin siis tulisi käyttää *ref*-avainsanaa? Sinun tulee käyttää *ref*-avainsanaa, kun sinun pitää varmistaa, että kutsuva metodi on alustanut parametrin. Yllä olevassa esimerkissä voitiin *out*-avainsanaa käyttää, koska kutsuttava metodi ei riippunut välitettävien parametrien arvoista. Mutta entä jos parametrin arvoa käytetään kutsutussa metodissa? Katsotaan seuraavaa koodia:

```
using System;

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

    protected int x;
    protected int y;

    public void Move(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public void ChangePos(ref int x, ref int y)
    {
        this.x += x;;
        this.y += y;

        x = this.x;
        y = this.y;
    }
}

class OutTest2App
```

(jatkuu)

```
{
    public static void Main()
    {
        Window wnd = new Window(5, 5);

        int x = 5;
        int y = 5;

        wnd.ChangePos(ref x, ref y);
        Console.WriteLine("{0}, {1}", x, y);

        x = -1;
        y = -1;
        wnd.ChangePos(ref x, ref y);
        Console.WriteLine("{0}, {1}", x, y);
    }
}
```

Kuten näet, kutsuttava metodi (*Window.ChangePos*) perustaa tekemisensä sille välitettyihin parametreihin. Tässä tapauksessa *ref*-avainsana pakottaa kutsuvan ohjelman alustamaan arvot, jotta metodi voisi toimia kunnolla.

Metodin ylikuormitus

Metodin ylikuormituksen avulla C#-ohjelmoija voi käyttää samaa metodin nimeä useita kertoja mutta erilaisella parametriluettelolla. Tämä on äärimmäisen käyttökelpoista kahdessa tilanteessa. Ensimmäinen koskee tilanteita, joissa haluat käyttää yhtä metodin nimeä monta kertaa, kun metodien toiminta eroaa hieman riippuen välitettävien parametrien tyypistä. Sanotaan esimerkiksi, että sinulla on kirjautumisluokka, jonka avulla sovelluksesi voi kirjoittaa tilastollista käyttöä varten tietoja levylle. Saadaksesi metodista joustavamman, sinulla voi olla *Write*-metodista erilaisia muotoja määrittelemään kirjoitettavaa informaatiota. Sen lisäksi, että hyväksyy parametrina kirjoitettavan merkkijonon, se voi myös hyväksyä resurssin tunnisteen kokonaislukuna. Ilman metodin ylikuormitusmahdollisuutta sinun pitäisi toteuttaa metodi molempia tilanteita varten, esimerkiksi *WriteString* ja *WriteFromResourceId*. Metodin ylikuormituksella voit kuitenkin toteuttaa seuraavat *WriteEntry*-nimiset metodit, jotka eroavat toisistaan vain parametrin tyypin verran:

```
using System;

class Log
{
    public Log(string fileName)
```

```
    {
        // Avaa fileName-nimisen  tiedoston.
    }

    public void WriteEntry(string entry)
    {
        Console.WriteLine(entry);
    }

    public void WriteEntry(int resourceId)
    {
        Console.WriteLine
       (“Retrieve string using resource id and write to log”);
    }
}

class Overloading1App
{
    public static void Main()
    {
        Log log = new Log(“My File”);
        log.WriteEntry(“Entry one”);
        log.WriteEntry(42);
    }
}
```

Toinen tilanne, jossa metodin ylikuormitus on tarpeellinen, on silloin, kun käytät muodostimia, jotka ovat itse asiassa objektin luonnin yhteydessä kutsuttavia metodeja. Sanotaan, että haluat tehdä luokan, joka voidaan muodostaa useammalla kuin yhdellä tavalla, esimerkiksi niin, että se saa parametrinaan joko tiedoton osoittimen (*int*) tai tiedoston nimen (*string*) avatakseen tiedoston. Koska C#:n säännöt määräävät, että luokan muodostimen nimi pitää olla luokan nimi, et voi yksinkertaisesti tehdä erinimistä muodostinta kutakin erilaista parametrin tyyppiä varten. Sen sijaan sinun tulee ylikuormittaa muodostin näin:

```
using System;

class File
{
}

class CommaDelimitedFile
{
    public CommaDelimitedFile(String fileName)
    {
        Console.WriteLine(“Constructed with a file name”);
    }
```

(jatkuu)

```
    public CommaDelimitedFile(File file)
    {
        Console.WriteLine("Constructed with a file object");
    }
}

class Overloading2App
{
    public static void Main()
    {
        File file = new File();
        CommaDelimitedFile file2 = new CommaDelimitedFile(file);
        CommaDelimitedFile file3 =
new CommaDelimitedFile("Some file name");
    }
}
```

Yksi tärkeä seikka muistaa metodin ylikuormituksessa on se, että kunkin saman nimisen metodin parametriluettelon tulee olla erilainen. Siksi seuraava koodi ei käänny, koska ainoa ero kahden *Overloading3App.Foo*-version välillä on paluuarvon tyypissä:

```
using System;

class Overloading3App
{
    void Foo(double input)
    {
        Console.WriteLine("Overloading3App.Foo(double)");
    }

    // VIRHE: Vain erilaiset paluuarvon tyypit. Ei käänny.
    double Foo(double input)
    {
        Console.WriteLine("Overloading3App.Foo(double)
                          (second version)");
    }

    public static void Main()
    {
        Overloading3App app = new Overloading3App();

        double i = 5;
        app.Foo(i);
    }
}
```

Muuttuva määrä parametreja

Joskus metodille välitettävien parametrien määrää ei tiedetä ennen kuin ohjelman suorituksen aikana. Sanotaan esimerkiksi, että haluat luokan, joka piirtää kuvaan viivan x- ja y-koordinaattiparien mukaan. Voit tehdä luokkaan sellaisen metodin, joka hyväksyy parametrinaan yhden *Point*-objektin, joka esittää sekä x- että y-arvoa. Tämä objekti voi sitten tallentaa kunkin *Point*-objektin linkitettyyn listaan tai muistitaulukkoon, kunnes kutsuva ohjelma haluaa tulostaa viivan pisteiden väliin. Tämä on kuitenkin huono suunnitelma muutaman syyn takia. Ensinnäkin se vaatii, että käyttää tekee tarpeetonta työtä kutsumalla metodia jokaisen piirrettävän viivan pistettä kohden (hyvin turhauttavaa, jos viiva on pitkä) ja sen jälkeen kutsuu toista metodia saadakseen viivan piirrettyä. Toinen huono puoli on se, että se vaatii luokan, joka tallentaa nämä pisteet jollakin tavalla, vaikka ainoa tarve pisteille on yhden metodin, *DrawLine*, käyttö. Muuttuva parametrien määrä on erinomainen tällaisen tilanteen ratkaisuun.

Voit määritellä muuttuvan määrän metodin parametreja käyttämällä *params*-avainsanaa ja määrittelemällä taulukon metodin parametriluetteloon. Seuraavassa esimerkki *Draw*-luokasta. Käyttäjä voi tehdän yhden *DrawLine*-metodin kutsun ja välittää sille haluamansa määrän *Point*-objekteja:

```
using System;

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

    public int x;
    public int y;
}

class Chart
{
    public void DrawLine(params Point[] p)
    {
        Console.WriteLine("\nThis method would print a line " +
                          "along the following points:");
        for (int i = 0; i < p.GetLength(0); i++)
```

(jatkuu)

```
        {
            Console.WriteLine("{0}, {1}", p[i].x, p[i].y);
        }
    }
}

class VarArgsApp
{
    public static void Main()
    {
        Point p1 = new Point(5,10);
        Point p2 = new Point(5, 15);
        Point p3 = new Point(5, 20);

        Chart chart = new Chart();
        chart.DrawLine(p1, p2, p3);
    }
}
```

DrawLine-metodi kertoo C#-kääntäjälle, että se voi ottaa vastaan muuttuvan määrän *Point*-objekteja. Suorituksen aikana metodi käyttää yksinkertaista for-silmukkaa käyden läpi välitetyt *Point*-objektit tulostaen ne.

Huomaa, että todellisessa sovelluksessa olisi parempi käyttää ominaisuuksia, joiden avulla käsitellä *Point*-objektin *x* ja *y* jäseniä sen sijaan, että tekee niistä *public*-tyyppisiä. Lisäksi olisi myös parempi käyttää *DrawLine*-metodissa *foreach*-käskyä *for*-silmukan sijasta. Tämän kirjan jatkuvuuden vuoksi en ole puhunut näistä mahdollisuuksista vielä. Puhun ominaisuuksista luvussa 7, "Ominaisuudet, taulukot ja indeksoijat," ja *foreach*-käskystä luvussa 11, "Ohjausrakenteet."

Virtuaaliset metodit

Kuten näimme luvussa 5, voit periyttää luokan toisesta niin, että luokka voi periä ja rakentua olemassa olevan luokan ominaisuuksille. Koska en ollut vielä silloin puhunut metodeista, puhuin silloin vain kenttien ja metodien periytymisestä. Toisin sanoen, en kertonut mahdollisuudesta muokata kantaluokan käyttäytymistä periytyvässä luokassa. Tämä tehdään käyttämällä virtuaalisia metodeja ja ne ovat tämä kappaleen aihe.

Metodin korvaaminen

Katsotaan ensin miten korvataan kantaluokan toiminnallisuus perityssä metodissa. Aloitamme kantaluokalla, joka esittää työntekijää. Pitääksemme esimerkin mahdollisimman

yksinkertaisena, teemme kantaluokkaan vain yhden *CalculatePay*-nimisen metodin, joka ei tee muuta kuin ilmoittaa kutsutun metodin nimen. Tämä auttaa meitä myöhemmin määrittämään, mitä periytymispuun metodia milloinkin kutsutaan.

```
class Employee
{
    public void CalculatePay()
    {
        Console.WriteLine("Employee.CalculatePay()");
    }
}
```

Sanotaan nyt, että haluat periyttää luokan *Employee*-luokasta ja haluat korvata *CalculatePay*-metodin ja saada sen tekemään periytyvässä luokassa jotain erikoista. Teet sen käyttämällä new-avainsanaa periytetyn luokan metodin märittelyssä. Tässä koodi, joka näyttää miten helppoa se on:

```
using System;

class Employee
{
    public void CalculatePay()
    {
        Console.WriteLine("Employee.CalculatePay()");
    }
}

class SalariedEmployee : Employee
{
    // new-avainsana mahdollistaa kantaluokan
    // toteutuksen korvaamisen.
    new public void CalculatePay()
    {
        Console.WriteLine("SalariedEmployee.CalculatePay()");
    }
}

class Poly1App
{
    public static void Main()
    {
        Poly1App poly1 = new Poly1App();

        Employee baseE = new Employee();
        baseE.CalculatePay();
```

(jatkuu)

```
        SalariedEmployee s = new SalariedEmployee();
        s.CalculatePay();
    }
}
```

Tämän sovelluksen kääntäminen ja suorittaminen aiheuttaa seuraavan tulosteen:

```
c:\>Poly1App
Employee.CalculatePay()
Salaried.CalculatePay()
```

Monimuotoisuus

Metodin korvaaminen *new*-avainsanalla toimii hyvin, jos sinulla on viittaus periytettyyn objektiin. Mutta mitä tapahtuu, jos sinulla on objektihierarkkiassa ylöspäin tyyppimuunnettu (upcasted) viittaus kantaluokkaan ja haluat kääntäjän kutsuvan metodista periytetyn luokan toteutusta ? Tässä tulee esiin monimuotoisuus. Sen avulla voit määritellä metodin useita kertoja läpi luokkahierarkkiasi siten, että ajonaikainen ympäristö kutsuu metodin oikeaa versiota riippuen käytettävästä objektista.

Katsotaan Employee-esimerkistä mitä tarkoitan. Poly1App-sovellus toimii oikein, koska meillä on kaksi objektia: *Employee* ja *SalariedEmployee*. Käytännönläheisemmässä esimerkissä lukisimme luultavasti kaikki työntekijätietueet tietokannasta ja täyttäisimme taulukon. Vaikka jotkin työntekijöistä olisivat urakoitsijoita ja jotkin kuukausipalkkaisia, meidän tulee sijoittaa ne kaikki yhtä tyyppiä olevaan taulukkoon, ja se tyyppi on kantaluokka, *Employee*. Kun sitten käymme läpi tätä taulukkoa ja kutsumme kunkin objektin *CalculatePay*-metodia, haluamme kääntäjän kutsuvan oikean objektin toteutusta *CalvculatePay*-metodista.

Seuraavassa esimerkissä olen lisännyt ohjelmaamme uuden luokan nimeltä *ContractEmployee*. Pääsovelluksen luokka sisältää nyt taulukon, joka on tyyppiä *Employee* ja kaksi lisämetodia, joista *LoadEmployee* lataa työntekijä-objektit taulukkoon ja *DoPayroll* käy taulukon läpi kutsuen kunkin objektin *CalculatePay*-metodia.

```
using System;

class Employee
{
    public Employee(string name)
    {
        this.Name = name;
    }

    protected string Name;
```

```
    public string name
    {
        get
        {
            return this.Name;
        }
    }

    public void CalculatePay()
    {
        Console.WriteLine("Employee.CalculatePay called for {0}",
                          name);
    }
}

class ContractEmployee : Employee
{
    public ContractEmployee(string name)
    : base(name)
    {
    }

    public new void CalculatePay()
    {
        Console.WriteLine("ContractEmployee.CalculatePay called for {0}",
                          name);
    }
}

class SalariedEmployee : Employee
{
    public SalariedEmployee (string name)
    : base(name)
    {
    }

    public new void CalculatePay()
    {
        Console.WriteLine("SalariedEmployee.CalculatePay called for {0}",
                          name);
    }
}

class Poly2App
{
    protected Employee[] employees;
```

(jatkuu)

```
    public void LoadEmployees()
    {
        // Simulating loading from a database.
        employees = new Employee[2];
        employees[0] = new ContractEmployee("Kate Dresen");
        employees[1] = new SalariedEmployee("Megan Sherman");
    }

    public void DoPayroll()
    {
        foreach(Employee emp in employees)
        {
            emp.CalculatePay();
        }
    }

    public static void Main()
    {
        Poly2App poly2 = new Poly2App();
        poly2.LoadEmployees();
        poly2.DoPayroll();
    }
}
```

Tämän sovelluksen suorittaminen saa aikaan seuraavat tulokset:

```
c:\>Poly2App
Employee.CalculatePay called for Kate Dresen
Employee.CalculatePay called for Megan Sherman
```

Tämä ei ole se, mitä halusimme, koska kunkin objektin kohdalla kutsutaan kantaluokan toteutusta *CalculatePay*-metodista. Se, mitä tapahtui, on esimerkki *aikaisesta sidonnasta* (early binding). Kun koodi käännettiin, C#-kääntäjä tutki *emp.CalculatePay*-kutsua ja päätteli sen muistipaikan osoitteen, jonne kutsun tapahtuessa pitää hypätä. Tässä tapauksessa se on *Employee.CalculatePay*-metodin sijaintipaikka.

Katsotaan seuraavaa MSIL-koodia, joka on generoitu Poly2App-sovelluksesta. Erityisen kiinnostava on sen rivi *IL_0014* ja se tosiasia, että sillä kutsutaan eksplisiittisesti *Employee.CalculatePay*-metodia.

```
.method public hidebysig instance void DoPayroll() il managed
{
  // Code size       34 (0x22)
  .maxstack  2
  .locals (class Employee V_0,
           class Employee[] V_1,
           int32 V_2,
           int32 V_3)
```

```
  IL_0000:  ldarg.0
  IL_0001:  ldfld      class Employee[] Poly2App::employees
  IL_0006:  stloc.1
  IL_0007:  ldloc.1
  IL_0008:  ldlen
  IL_0009:  conv.i4
  IL_000a:  stloc.2
  IL_000b:  ldc.i4.0
  IL_000c:  stloc.3
  IL_000d:  br.s       IL_001d
  IL_000f:  ldloc.1
  IL_0010:  ldloc.3
  IL_0011:  ldelem.ref
  IL_0012:  stloc.0
  IL_0013:  ldloc.0
  IL_0014:  call       instance void Employee::CalculatePay()
  IL_0019:  ldloc.3
  IL_001a:  ldc.i4.1
  IL_001b:  add
  IL_001c:  stloc.3
  IL_001d:  ldloc.3
  IL_001e:  ldloc.2
  IL_001f:  blt.s      IL_000f
  IL_0021:  ret
} // end of method Poly2App::DoPayroll
```

Tuo *Employee.CalculatePay*-metodin kutsu on ongelma. Tässä tapauksessa haluamme *myöhäisen sidonnan* (late binding). Se tarkoittaa, että kääntäjä ei valitse suoritettavaa metodia ennen kuin suorituksen aikana. Pakottaaksemme kääntäjän kutsumaan oikeaa varsiota ylöspäin tyyppimuunnetun (upcasted) objektin metodista, käytämme kahta uutta avainsanaa: *virtual* ja *override*. *virtual*-avainsanaa tulee käyttää kantaluokan metodissa ja *override*-avainsanaa käytetään metodin toteutuksessa periytyvässä luokassa. Tässä esimerkki uudelleen, tällä kertaa oikein toimivana!

```
using System;

class Employee
{
    public Employee(string name)
    {
        this.Name = name;
    }

    protected string Name;
    public string name
```

(jatkuu)

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

        virtual public void CalculatePay()
        {
            Console.WriteLine(“Employee.CalculatePay called for {0}",
                              name);
        }
    }

    class ContractEmployee : Employee
    {
        public ContractEmployee(string name)
        : base(name)
        {
        }
        override public void CalculatePay()
        {
            Console.WriteLine(“ContractEmployee.CalculatePay called for {0}",
                              name);
        }
    }

    class SalariedEmployee : Employee
    {
        public SalariedEmployee (string name)
        : base(name)
        {
        }
        override public void CalculatePay()
        {
            Console.WriteLine(“SalariedEmployee.CalculatePay called for {0}",
                              name);
        }
    }

    class Poly3App
    {
        protected Employee[] employees;
        public void LoadEmployees()
        {
            // Simulating loading from a database.
            employees = new Employee[2];
```

```
        employees[0] = new ContractEmployee("Kate Dresen");
        employees[1] = new SalariedEmployee("Megan Sherman");
    }

    public void DoPayroll()
    {
        foreach(Employee emp in employees)
        {
            emp.CalculatePay();
        }
    }

    public static void Main()
    {
        Poly3App poly3 = new Poly3App();
        poly3.LoadEmployees();
        poly3.DoPayroll();
    }
}
```

Ennen kuin käynnistämme sovelluksen, vilkaistaanpa generoitua IL-koodia. Tällä kertaa huomaamme, että rivi *IL_0014* käyttää MSIL:n käskyä *callvirt*, joka kertoo kääntäjälle, että kutsuttavaa metodia ei tiedetä täsmällisesti ennen kuin suorituksen aikana, koska se riippuu siitä, mitä periytyvää objektia käytetään:

```
.method public hidebysig instance void DoPayroll() il managed
{
  // Code size       34 (0x22)
  .maxstack  2
  .locals (class Employee V_0,
           class Employee[] V_1,
           int32 V_2,
           int32 V_3)
  IL_0000:  ldarg.0
  IL_0001:  ldfld      class Employee[] Poly3App::employees
  IL_0006:  stloc.1
  IL_0007:  ldloc.1
  IL_0008:  ldlen
  IL_0009:  conv.i4
  IL_000a:  stloc.2
  IL_000b:  ldc.i4.0
  IL_000c:  stloc.3
  IL_000d:  br.s       IL_001d
  IL_000f:  ldloc.1
  IL_0010:  ldloc.3
  IL_0011:  ldelem.ref
  IL_0012:  stloc.0
  IL_0013:  ldloc.0
```

(jatkuu)

```
  IL_0014:  callvirt   instance void Employee::CalculatePay()
  IL_0019:  ldloc.3
  IL_001a:  ldc.i4.1
  IL_001b:  add
  IL_001c:  stloc.3
  IL_001d:  ldloc.3
  IL_001e:  ldloc.2
  IL_001f:  blt.s      IL_000f
  IL_0021:  ret
} // end of method Poly3App::DoPayroll
```

Koodin suorittaminen johtaa nyt seuraaviin tulksiin:

```
c:\>Poly3App
ContractEmployee.CalculatePay called for Kate Dresen
SalariedEmployee.CalculatePay called for Megan Sherman
```

> Huomaa Virtuaalinen metodi ei voi olla määritelty määreellä *private*, koska se määrityksensä mukaan ei saa olla näkyvä periytyvissä luokissa.

Staattiset metodit

Staattinen metodi on olemassa luokalla kokonaisuutena eikä sen määrätyllä instanssilla. Kuten muutkin staattiset jäsenet, staattisen metodin perusominaisuus on se, että se sijaitsee erillään luokan instansseista sovelluksen yleisessä muistissa ja sitä ei, vastoin oliohjelmoinnin yleisiä periaatteita, liitetä yhteenkään instanssiin. Esimerkkinä tästä olkoon tietokantasovellus, jonka kirjoitin C#:lla. Luokkahierarkkiassani minulla on luokka nimeltä *SQLServerDb*. Perusominaisuuksien (lisää, päivitä, lue ja poista) lisäksi luokassa on myös tietokannan korjausmetodi. Luokan *Repair*-metodissa minun ei tarvitse avata varsinaista tietokantaa. Itse asiassa käyttämäni ODBC-funktio (*SQLConfigDataSource*) edellyttää, että tietokanta on suljettu toiminnon ajan. *SQLServerDb*-luokan muodostin avaa sille välitetyn nimen mukaisen tietokannan. Siksi staattinen metodi sopii tähän. Se mahdollistaa, että voin sijoittaa metodin *SQLServerDb*-luokkaan, jonne se kuuluu eikä luokkani muodostimeen. Asiakasohjelman kannalta etu on se, että sen ei tarvitse luoda instanssia myöskään *SQLServerDb*-luokasta. Seuraavassa esimerkissä näet, että staattista metodia (*RepairDatabase*) kutsutaan *Main*-metodista. Huomaa, että *SQLServerDB*-luokasta ei tarvitse luoda instanssia:

```
using System;

class SQLServerDb
{
    // Joukko näkymättömiä metodeja
    public static void RepairDatabase()
    {
        Console.WriteLine(“repairing database...”);
    }
}

class StaticMethod1App
{
    public static void Main()
    {
        SQLServerDb.RepairDatabase();
    }
}
```

Metodin määrittäminen staattiseksi onnistuu *static*-avainsanalla. Sitten käyttäjä käyttää *Luokka.Metodi*-rakennetta kutsuakseen sitä. Huomaa, että tätä rakennetta on käytettävä, vaikka käyttäjällä on viittaus luokan instanssiin. Tämän osoittaa seuraava esimerkki, joka ei käänny:

```
// Tämä koodi ei käänny.
using System;

class SQLServerDb
{
    // Joukko näkymättömiä metodeja.
    public static void RepairDatabase()
    {
        Console.WriteLine(“repairing database...”);
    }
}

class StaticMethod2App
{
    public static void Main()
    {
        SQLServerDb db = new SQLServerDb();
        db.RepairDatabase();
    }
}
```

Luokan jäsenten käsittely

Viimeinen stattisiin metodeihin liittyvä sääntö, jonka käymme läpi, kertoo, mitä luokan jäseniä voidaan käsitellä staattisesta metodista. Kuten ehkä arvaatkin, staattinen metodi voi käsitellä luokan jokaista staattista jäsentä mutta ei instanssijäseniä. Seuraava koodi selventää tätä asiaa:

```
using System;

class SQLServerDb
{
    static string progressString1 = “repairing database...";
    string progressString2 = “repairing database...";

    public static void RepairDatabase()
    {
        Console.WriteLine(progressString1); // Tämä  toimii.
        Console.WriteLine(progressString2); //  Ei  käänny.
    }
}

class StaticMethod3App
{
    public static void Main()
    {
        SQLServerDb.RepairDatabase();
    }
}
```

Yhteenveto

Metodit antavat luokille niiden toiminnalliset ominaisuudet ja suorittavat asioita puolestamme. Metodit ovat C#:ssa joustavia, mahdollistaen useiden arvojen palauttamisen, ylikuormituksen ja muuttuvan määrän parametreja. *ref* ja *out*-avainsanojen avulla metodi voi palauttaa useamman kuin yhden arvon kutsujalle. Ylikuormitus mahdollistaa useiden saman nimisten metodien toimivan eri tavalla riippuen niille välitettavien parametrien tyypistä ja/tai määrästä. Metodeille voidaan välittää muuttuva määrä parametreja. *params*-avainsanalla voit käsitellä metodeja, joiden parametrien määrää ei tiedetä ennen kuin suorituksen aikana. Virtuaalisen metodin avulla voit määrätä, miten metodeja muokataan periytyvissä luokissa. *static*-avainsanalla voit määritellä metodeja, jotka ovat olemassa luokan eivätkä yksittäisen instanssin osana.