7

Ominaisuudet, taulukot ja indeksoijat

Tähän mennessä olen käsitellyt C#:n tukemia perustyyppejä ja sitä, miten niitä määritellään ja käytetään luokissa ja sovelluksissa. Tämä luku rikkoo periaatteen, jonka mukaan yksi luku sisältää yhden kielen peruspiirteen esittelyn. Tässä luvussa opit ominaisuudet, taulukot ja indeksoijat, koska ne kuuluvat yhteen. Niiden avulla sinä voit laajentaa kielen luokka/kenttä/metodi-perusrakennetta siten, että voit tarjota luonnollisemman rajapinnan luokkasi käyttäjille.

Ominaisuudet ovat älykkäitä kenttiä

On aina hyvä tavoite suunnitella luokat niin, että ne eivät pelkästään piilota metodien toteutusta vaan myös estävät luokan kenttien suoran käsittelyn. *Käsittelymetodien* (accessor method) avulla voit lukea ja asettaa kenttien arvoja ja samalla varmistaa, että kenttää käsitellään oikein eli sovelluksesi liiketoimintasääntöjen mukaan. Samalla voit helposti käsitellä kenttien arvoja tarvittavalla tavalla.

Kuvitellaan, että sinulla on osoiteluokka (Address), jossa on kentät ZIP-koodille (ZipCode) ja kaupungin nimelle (City). Kun asiakasohjelma muokkaa *Address.ZipCode*-kenttää, haluat tarkistaa sen kelpoisuuden tietokannasta ja täyttää sen perusteella automaattisesti *Address.City*-kentän. Jos asiakasohjelmalla olisi suora pääsy *Address.ZipCode*-jäseneen, et voisi helposti tehdä kumpaakaan asiaa, koska jäsenen suoraan muuttamiseen ei tarvita metodia. Parempi vaihtoehto kuin antaa suora käsittelyoikeus *Address.ZipCode*-kenttään, on määritellä *Address.City* ja *Address.ZipCode*-kentät määreellä *protected* ja toteuttaa käsittelymetodit *Address.ZipCode*-kentän lukemista ja asettamista varten. Tällä tavalla voit liittää tietokantatarkistukseen ja kaupungin nimen hakemiseen tarvittivan koodin *ZipCode*-kentän käsittelymetodiin

Tämä esimerkki koodataan C#:lla seuraavasti. Huomaa, että *ZipCode*-kenttä määritellään määreellä *protected*, jotta sitä ei voi käsitellä asiakasohjelmassa suoraan, ja käsittelymetodit *GetZipCode* ja *SetZipCode* määreellä *public*.

```
class Address
{
    protected string ZipCode;
    protected string City;
    public string GetZipCode()
    {
        return this.ZipCode;
    }
    public void SetZipCode(string ZipCode)
    {
        // Tarkistetaan arvon kelpoisuus.
        this.ZipCode = ZipCode;
        // Päivitetään city zipCode:n  perusteella.
    }
}
```

Asiakasohjelma voi käsitellä *Address.ZipCode*-arvoa seuraavasti:

```
Address addr = new Address();
addr.SetZipCode("55555");
string zip = addr.GetZipCode();
```

Ominaisuuksien määrittely ja käyttäminen

Käsittelymetodien käyttäminen toimii hyvin ja se on tekniikka, jota käytetää muissakin olioperusteisissa kielissä, kuten C++:ssa ja Javassa. C#:ssa on kuitenkin vieläkin monipuolisempi menetelmä, ominaisuudet, joilla on samat ominaisuudet kuin käsittelymetodeilla mutta käyttäjän kannalta katsottuna ne ovat tyylikkäämpiä. Ominaisuuksien avulla ohjelmoija voi kirjoittaa asiakasohjelman, joka voi käsitellä luokan kenttiä aivan kuin ne olisivat *public*-tyyppisiä, ilman että hänen pitää tietää, onko käsittelymetodeja olemassa.

C#-ominaisuus koostuu kentän määrittelystä ja käsittelymetodeista, joilla kentän arvoa muokataan. Nämä käsittelymetodit ovat nimeltään *getter* ja *setter*-metodit. Getter-metodeita käytetään kentän arvon hakemiseen ja setter-metodeita kentän arvon asettamiseen. Tässä edellä ollut esimerkki uudelleen kirjoitettuna nyt hyödyntäen C#:n ominaisuuksia:

```
class Address
{
    protected string city;
    protected string zipCode;
```

```
        public string ZipCode
        {
            get
            {
                return zipCode;
            }
            set
            {
                // Tarkistetaan arvon kelpoisuus.
                zipCode = value;
                // Päivitetään city zipCode:n  perusteella.
            }
        }
    }
```

Huomaa, että loin kentän nimeltä *Address.zipCode* ja ominaisuuden nimeltä *Address.ZipCode*. Tämä voi aiheuttaa joillekin aluksi sekaannusta, koska he luulevat, että *Address.ZipCode* on kenttä ja ihmettelevät, miksi se pitää määritellä kahdesti. Mutta se ei ole kenttä. Se on ominaisuus, joka on yleinen tapa määritellä käsittelijät luokan jäsenelle niin, että voidaan käyttää luonnollista merkintätapaa *objekti.kenttä*. Jos tässä esimerkissä olisin jättänyt pois *Address.zipCode*-kentän ja muuttanut setterin käskyn muodosta *zipCode = value* muotoon *ZipCode = value*, olisin aiheuttanut setter-metodin kutsumisen jatkuvasti. Huomaa myös, että setter-metodi ei ota parametreja. Välitettävä arvo sijoitetaan automaattisesti muuttujaan nimeltä value, jota voidaan käsitellä setter-metodin sisällä. (Näet kohta MSIL:ssä miten tämä taikuus tapahtuu.)

Nyt, kun olemme kirjoittaneet *Address.ZipCode*-ominaisuuden, katsotaanpa mitä muutoksia asiakasohjelman koodiin pitää tehdä:

```
Address addr = new Address();
addr.ZipCode = "55555";
string zip = addr.ZipCode;
```

Kuten näet, asiakasohjelma käsittelee kenttiä luonnollisesti: ei tarvita enää arvauksia tai dokumenttien tutkimista (eikä lähdekoodin) sen selvittämiseksi, onko kenttä *public*, ja jos jos ei ole, niin sen käsittelijämetodin nimen selvittämistä.

Mitä kääntäjä itse asiassa tekee

Miten kääntäjä antaa meidän kutsua metodia *objekti.kenttä*-rakenteella? Ja mistä *value*-muuttuja tulee? Saadaksemme vastauksen näihin kysymyksiin meidän pitää tutkia kääntäjän tuottamaa MSIL-koodia. Selvitetään ensin ominaisuuden getter-metodi.

Menossa olevassa esimerkissämme meillä on määriteltynä seuraavanlainen getter-metodi:

```
class Address
{
    protected string city;
    protected string zipCode;
    public string ZipCode
    {
        get
        {
            return zipCode;
        }
        §
    }
    §
}
```

Jos katsot tämän metodin MSIL-koodia, huomaat, että kääntäjä on luonut käsittelymetodin nimeltä *get_ZipCode*:

```
.method public hidebysig specialname instance string
        get_ZipCode() cil managed
{
  // Code size       11 (0xb)
  .maxstack  1
  .locals ([0] string _Vb_t_$00000003$00000000)
  IL_0000:  ldarg.0
  IL_0001:  ldfld      string Address::zipCode
  IL_0006:  stloc.0
  IL_0007:  br.s       IL_0009
  IL_0009:  ldloc.0
  IL_000a:  ret
} // end of method Address::get_ZipCode
```

Voit selvittää käsittelymetodin nimen, koska kääntäjä lisää ominaisuuden nimen alkuun merkit *get_* (getter-metodilla) tai *set_* (setter-metodilla). Lopputuloksena seuraava koodi ratkaisee kutsun *get_ZipCode*-metodiin:

```
String str = addr.ZipCode; // tämä kutsuu metodia
Address::get_ZipCode
```

Siksi voit yrittää kokeilla seuraava käsittelymetodin eksplisiittistä kutsua :

```
String str = addr.get_ZipCode; // **VIRHE – Ei käänny!
```

Tässä tapauksessa koodi ei kuitenkaan käänny, koska on kiellettyä kutsua suoraan MSIL:n sisäistä metodia.

Vastaus kysymyksemme (miten kääntäjä antaa meidän käyttää *objekti.kenttä*-rakennetta ja kuitenki kutsuu metodia?) kuuluu näin: kääntäjä todellisuudessa generoi vastaavat getter- ja setter-metodit puolestamme, kun se muodostaa C#:n ominaisuutta. Siksi, kun on kyse *Address.ZipCode*-ominaisuudesta, kääntäjä tuottaa MSIL:n, joka sisältää *get_ZipCode* ja *set_ZipCode*-metodit.

Katsotaan nyt generoitua setter-metodia. Näet *Address*-luokassa seuraavaa:

```
public string ZipCode
{
    §
    set
    {
        // Tarkistetaan arvon kelpoisuus.
        zipCode = value;
       // Päivitetään city zipCode:n perusteella.
    }
}
```

Huomaa, että mikään tässä koodissa ei määrittele muuttujaa nimeltä *value,* mutta silti voimme käyttää sitä kutsujan välittämän arvon tallennuspaikkana ja asettaa sen avulla *zipCode*-jäsenen arvon. Kun C#-kääntäjä generoi setter-metodin MSIL:n, se lisäsi tämän muuttujan parametriksi metodiin nimeltä *set_ZipCode*.

Generoidussa MSIL:ssä tämä metodi ottaa parametrikseen tyyppiä string olevan muuttujan:

```
.method public hidebysig specialname instance void
        set_ZipCode(string ‘value’) cil managed
{
  // Code size       8 (0x8)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  stfld      string Address::zipCode
  IL_0007:  ret
} // end of method Address::set_ZipCode
```

Vaikka et voikaan nähdä tätä metodia C#-lähdekoodissa, voit asettaa *ZipCode*-minaisuuden esimerkiksi näin *addr.ZipCode("12345")*, joka muutetaan MSIL-kutsuksi *Address::set_ZipCode("12345")*. Kuten *get_ZipCode*-metodin kohdallakin, yritys kutsua tätä metodia suoraan C#-koodissa aiheuttaa virheen.

Vain-luku-ominaisuudet

Käyttämässämme esimerkissä *Address.ZipCode*-ominaisuus on luku/kirjoitus-tyyppinen, koska sekä getter- että setter-metodi on määritelty. On tietenkin tilanteita, jolloin et halua, että asiakasohjelma voi asettaa määrätyn kentän arvon, jolloin teet kentästä vain-luku-tyyppisen. Teet sen jättämällä setter-metodin pois. Estetään esimerkin vuoksi asiakasohjelmaa asettamasta *Address.city*-kenttää ja jätetään *Address.ZipCode*-ominaisuus ainoaksi mahdollisuudeksi koodin avulla muuttaa sen arvoa:

```
class Address
{
    protected string city;
    public string City
    {
        get
        {
            return city;
        }
    }

    protected string zipCode;
    public string ZipCode
    {
        get
        {
            return zipCode;
        }
        set
        {
            // Tarkistetaan arvon kelpoisuus.
            zipCode = value;
            // Päivitetään city zipCode:n  perusteella.
        }
    }
}
```

Periytyvät ominaisuudet

Metodien tavoin ominaisuus voidaan varustaa *virtual*, *override* tai *abstract* määreellä, joista puhuin luvussa 6, "Metodit." Täten periytyvä luokka voi periä ja korvata ominaisuudet aivan samalla tavalla kuin minkä tahansa muun kantaluokan jäsenen. Oleellista on, että voi määritellä nämä määreet vain ominaisuuden tasolla. Toisin sanoen, kun sinulla on sekä getter- että setter-metodi, sinun pitää korvata ne molemmat eikä vain toista.

Ominaisuuksien erikoiskäyttö

Tähän mennessä olemme puhuneet ominaisuuksien käyttökelpoisuudesta seuraavista syistä:

- Ne tarjoavat abstraktiotason asiakasohjelmille.
- Ne tarjoavat yleisen käsittelytavan luokan jäsenille *objekti.kenttä*-rakenteen ansiosta.
- Niiden avulla luokka voi taata, että kaikki tarpeelliset toimet tehdään, kun määrättyä kenttää muokataan tai käsitellään.

Kolmas kohta mahdollistaa uuden hyödyllisen ominaisuuden käyttämisen: *laiskan alustuksen* (lazy initialization) toteuttaminen. Se on optimointitekniikka, jossa joitakin luokan jäseniä ei alusteta ennen kuin niitä tarvitaan.

Laiska alustus on hyödyllinen, kun sinulla on luokka, joka sisältää harvoin tarvittavia jäseniä, joiden alustaminen kestää kauan tai jotka tarvitsevat paljon resursseja. Esimerkkinä voisi olla tilanne, jossa pitää lukea tietoja tietokannasta tai ruuhkaisen verkon yli. Koska tiedät, että näihin jäseniin viitataan harvoin ja niiden alustaminen on raskas toimenpide, voit viivyttää alustusta, kunnes niiden getter-metodia kutsutaan. Sanotaan, että sinulla on varastosovellus, jota myyntiedustajat kannettavilla koneillaan käyttävät syöttäessään asiakkaiden tilauksia ja joskus tarkistaessaan varastosaldoja. Ominaisuuksia käyttämällä voit sallia luokan instantioinnin ilman, että varastosaldoja pitää lukea, kuten näet alla olevassa koodissa. Jos myyntiedustaja sitten haluaa käsitellä tuotteen saldoa, getter-metodi ottaa yhteyden tietokantaan ja tekee tarpeellisen kyselyn.

```
class Sku
{
    protected double onHand;

    public string OnHand
    {
        get
        {
            // Luetaan keskuskoneelta ja asetetaan arvo.
            return onHand;
        }
    }
}
```

Kuten olet tässä luvussa nähnyt, voit ominaisuuksien avulla tarjota kentille käsittelymetodit ja asiakasohjelmalle yleisen ja helppokäyttöisen rajapinnan. Tästä syystä ominaisuuksia sanotaan joskus *älykkäiksi kentiksi* (smart field). Otetaan nyt askel

pidemmälle ja katsotaan, miten taulukkoja määritellään ja käytetään C#:ssa. Näemme myös miten ominaisuuksia käytetään taulukon kanssa, jolloin saadaan aikaan indeksoija.

Taulukot

Tähän mennessä useimmat kirjan esimerkit ovat määritelleet rajallisen, ennaltamäärätyn määrän muuttujia. Useimmissa todellisissa sovelluksissa et kuitenkaan tiedä objektien täsmällistä määrää ennen kuin suorituksen aikana. Jos olet esimerkiksi tekemässä editoria ja haluat pitää kirjaa valintaikkunaan lisätyistä kontrolleista, et tiedä editorin näyttämien kontrollien määrää ennen kuin ohjelma on käynnissä. Voit kuitenkin käyttää taulukkoa tallentamaan dynaamisesti varattuja objektien joukkoa.

C#:ssa taulukot ovat objekteja, joiden kantaluokka on *System.Array*. Vaikka taulukon määrittelyn syntaksi näyttääkin samanlaiselta kuin C++:ssa tai Javassa, luot itse asiassa instanssin .NET-luokasta, joka tarkoittaa, että jokaisella taulukolla on kaikki samat jäsenet perittynä *System.Array*-luokasta. Tässä kappaleessa kerron, miten taulukoita määritellään ja instantioidaan, miten käsitellään eri tyyppisiä taulukoita ja miten käydään läpi taulukon elementit. Tarkastelen myös muutamia yleisimmin käytettyjä *System.Array*-luokan ominaisuuksia ja metodeja.

Taulukon määrittely

Taulukko määritellään C#:ssa kirjoittamalla tyhjät hakasulkeet tyypin ja muuttujan nimen väliin, näin:

```
int[] numbers;
```

Huomaa, että tämä eroaa hieman C++:n määrittelystä, jossa hakasulkeet sijoitetaan muuttujan nimen perään. Koska taulukot ovat luokkaan perustuvia, pätevät niiden määrittelyyn monet samat säännöt kuin luokan määrittelyyn. Kun esimerkiksi määrittelet taulukon, et itse asiassa luo taulukkoa. Aivan kuten luokan kohdalla, sinun tulee instantioida taulukko ennen kuin sen elementeille varataan tilaa. Seuraavassa esimerkissä määrittelen ja instantioin luokan samalla kertaa:

```
// Määritellään ja instantioidaan yksiulotteinen
// taulukko kuudelle kokonaisluvulle
int[] numbers = new int[6];
```

Kuitenkin, kun määrittelet taulukon luokan jäseneksi, sinun tulee määritellä ja instantioida taulukko kahdessa vaiheessa, koska et voi instantioida objektia ennen kuin suorituksen aikana.

```
class YourClass
{
    §
    int[] numbers;
    §

    void SomeInitMethod()
    {
        §
        numbers = new int[6];
        §
    }
}
```

Esimerkki yksiulotteisen taulukon käytöstä

Tässä yksinkertainen esimerkki yksiulotteisen taulukon määrittelemisestä luokan jäseneksi, sen instantioinnista ja täyttämisestä muodostimessa ja sen jälkeen for-silmukan käyttämisestä sen läpikäyntiin ja kunkin elementin tulostamiseen.

```
using System;

class SingleDimArrayApp
{
    protected int[] numbers;

    SingleDimArrayApp()
    {
        numbers = new int[6];
        for (int i = 0; i < 6; i++)
        {
            numbers[i] = i * i;
        }
    }

    protected void PrintArray()
    {
        for (int i = 0; i < numbers.Length; i++)
        {
            Console.WriteLine("numbers[{0}]={1}", i, numbers[i]);
        }
    }
```

(jatkuu)

```
    public static void Main()
    {
        SingleDimArrayApp app = new SingleDimArrayApp();
        app.PrintArray();
    }
}
```

Esimerkin suorittaminen antaa seuraavan tuloksen:

```
numbers[0]=0
numbers[1]=1
numbers[2]=4
numbers[3]=9
numbers[4]=16
numbers[5]=25
```

Tässä esimerkissä *SingleDimArray.PrintArray*-metodi käyttää *System.ArrayLength*-ominaisuutta määritellessään taulukon elementtien määrän. Koska meillä on tässä vain yksiulotteinen taulukko, tulee huomioida, että *Length*-ominaisuus todellisuudessa palauttaa taulukon kaikkien ulottuvuuksien elementtien yhteismäärän. Jos meillä olisi esimerkiksi kaksiulotteinen taulukko, jonka elementtien määrä toiseen ulottuvuuteen olisi 4 ja toiseen 5, *Length*-ominaisuus palauttaisi arvon 20. Seuraavassa kappaleessa tarkastelen moniulotteisia taulukoita ja sitä, miten määritellään taulukon määrätyn ulottuvuuden yläraja.

Moniulotteiset taulukot

Yksiulotteisen taulukon lisäksi C# tukee moniulotteisen taulukon määrittelyä. Tällaisen taulukon määrittelyssä kukin ulottuvuus erotetaan pilkulla. Seuraavassa määrittelen kolmiulotteisen double-tyyppisen taulukon:

```
double[,,] numbers;
```

Kun haluat nopeasti selvittää määritellyn C#-taulukon ulottuvuuksien määrän, laske pilkut ja vähennä yksi.

Seuraavassa esimerkissä minulla on kaksiulotteinen taulukko myyntitietoja varten. Taulukko sisältää tämän vuoden kuukausikohtaiset tiedot ja vastaavat tiedot viime vuodelta. Huomaa erityisesti taulukon instantioinnissa käytettävä syntaksi (*MultiDimArrayApp*:n muodostimessa.)

```
using System;

class MultiDimArrayApp
{
    protected int currentMonth;
    protected double[,] sales;

    MultiDimArrayApp()
    {
```

```
        currentMonth=10;

        sales = new double[2, currentMonth];
        for (int i = 0; i < sales.GetLength(0); i++)
        {
            for (int j=0; j < 10; j++)
            {
                sales[i,j] = (i * 100) + j;
            }
        }
    }

    protected void PrintSales()
    {
        for (int i = 0; i < sales.GetLength(0); i++)
        {
            for (int j=0; j < sales.GetLength(1); j++)
            {
                Console.WriteLine("[{0}][{1}]={2}", i, j, sales[i,j]);
            }
        }
    }

    public static void Main()
    {
        MultiDimArrayApp app = new MultiDimArrayApp();
        app.PrintSales();
    }
}
```

MultiDimArrayApp-esimerkin suorittaminen aiheuttaa tällaisen tulosteen:

```
[0][0]=0
[0][1]=1
[0][2]=2
[0][3]=3
[0][4]=4
[0][5]=5
[0][6]=6
[0][7]=7
[0][8]=8
[0][9]=9
[1][0]=100
[1][1]=101
[1][2]=102
[1][3]=103
[1][4]=104
[1][5]=105
```

(jatkuu)

```
[1][6]=106
[1][7]=107
[1][8]=108
[1][9]=109
```

Muistathan, miten yksiulotteisen taulukon esimerkin yhteydessä mainitsin, että *Lenght*-ominaisuus palauttaa taulukon elementtien kokonaismäärän, joten tässä esimerkissä sen paluuarvo olisi 20. *MultiDimArray.PrintSales*-metodissa käytin *Array.GetLength*-metodia määritelläkseni taulukon kunkin ulottuvuuden pituuden, tai oikeastaan sen ylärajan. Sen jälkeen pystyin käyttämään sitä *PrintSales*-metodissa.

Ulottuvuuksien määrän selvittäminen

Kun nyt olemme nähneet, miten helppoa on käydä läpi yksi- tai moniulotteinen taulukko, saatat miettiä, miten voit selvittää taulukon ulottuvuuksien määrän ohjelmallisesti. Taulukon ulottuvuuksien määrään kutsutaan englanninkielisellä termillä *rank* ja sen arvon saat selville käyttämällä *Array.Rank*-ominaisuutta. Tässä muutaman taulukon verran esimerkkejä sen käytöstä :

```
using System;

class RankArrayApp
{
    int[] singleD;
    int[,] doubleD;
    int[,,] tripleD;

    protected RankArrayApp()
    {
        singleD = new int[6];
        doubleD = new int[6,7];
        tripleD = new int[6,7,8];
    }

    protected void PrintRanks()
    {

          Console.WriteLine("singleD Rank = {0}", singleD.Rank);
        Console.WriteLine("doubleD Rank = {0}", doubleD.Rank);
        Console.WriteLine("tripleD Rank = {0}", tripleD.Rank);
    }

    public static void Main()
    {
        RankArrayApp app = new RankArrayApp();
        app.PrintRanks();
    }
}
```

Odotusten mukaisesti *RankArrayApp*-sovellus antaa seuraavat tulokset:

```
singleD Rank = 1
doubleD Rank = 2
tripleD Rank = 3
```

Sisäkkäiset taulukot

Viimeinen asia, jota taulukoista tutkimme, on *sisäkkäiset taulukot* (jagged arrays). Se on yksinkertaisesti taulukko taulukoita. Tässä esimerkki kokonaislukutaulukoita sisältävän taulukon määrittelystä:

```
int[][] jaggedArray;
```

Voit käyttää sisäkkäistä taulukkoa, jos olet tekemässä editoria. Siinä sinun pitää tallentaa kutakin käyttäjän luomaa kontrollia esittävä objekti taulukkoon. Sanotaan, että sinulla on taulukko painikkeita (button) ja avautuvia luetteloja (combo box). Sinulla on esimerkiksi kolme painiketta ja kaksi luetteloa, jotka on molemmat tallennettu omaan taulukkoonsa. Määrittelemällä sisäkkäisen taulukon saat niille "isätaulukon" ja voit tarvittaessa helposti käydä kontrollit ohjelmallisesti läpi, kuten näet seuraavasta esimerkistä:

```
using System;

class Control
{
    virtual public void SayHi()
    {
        Console.WriteLine("base control class");
    }
}

class Button : Control
{
    override public void SayHi()
    {
        Console.WriteLine("button control");
    }
}

class Combo : Control
{
    override public void SayHi()
    {
        Console.WriteLine("combobox control");
    }
}
class JaggedArrayApp
{
    public static void Main()
```

(jatkuu)

```
    {
    Control[][] controls;
    controls = new Control[2][];

    controls[0] = new Control[3];
    for (int i = 0; i < controls[0].Length; i++)
    {
        controls[0][i] = new Button();
    }

    controls[1] = new Control[2];
    for (int i = 0; i < controls[1].Length; i++)
    {
        controls[1][i] = new Combo();
    }

    for (int i = 0; i < controls.Length;i++)
    {
        for (int j=0;j< controls[i].Length;j++)
        {
            Control control = controls[i][j];
            control.SayHi();
        }
    }

    string str = Console.ReadLine();
    }
}
```

Kuten näet, määrittelin kantaluokan (*Control*) ja kaksi periytyvää luokkaa (*Button* ja *Combo*) ja määrittelin sisäkkäisen taulukon siten, että se sisältää *Controls*-objekteja sisältäviä taulukoita. Tällä tavoin voin tallentaa taulukkoon määrätyn tyyppisiä objekteja, ja monimuotoisuuden ansiosta tiedän, että kun on aika hakea objektit taulukosta, saan odottamani toiminnan.

Objektien käsitteleminen taulukon tavoin indeksoijien avulla

"Taulukot"-kappaleessa opit, miten määritellään ja instantioidaan taulukoita, miten eri taulukkotyyppejä käytetään ja miten käydään läpi taulukon elementit. Opit myös miten hyödynnät yleisimmin käytettyjä *System.Array*-luokasta periytyviä taulukoiden ominaisuuksia ja metodeja. Jatketaan taulukoiden parissa tutkimalla, miten C#-ominaisuuden nimeltä *indeksoija* avulla voit käsitellä objektia ohjelmallisesti aivan kuin se olisi taulukko.

Miksi haluaisit käsitellä objektia kuin taulukkoa? Kuten useimpien ohjelmointikielen ominaisuuksien, myös indeksoijan käytön edut tulevat siitä, että sovelluksen tekeminen helpottuu. Tämä luvun ensimmäisessä kappaleessa, "Ominaisuudet ovat älykkäitä kenttiä," näit miten C#:n ominaisuuksien avulla voit viitata luokan jäseneen *luokka.kenttä*-rakenteen avulla, vaikka tällöin todellisuudessa käytetäänkin getter- ja setter-metodeja. Tämä abstraktio vapauttaa ohjelmoijan tekemään luokkaa käyttävän ohjelman tutkimatta, onko kentällä olemassa getter/setter-metodit ja minkälaisia ne täsmälleen ovat. Vastaavasti indeksoijat mahdollistavat luokkaa käyttävän ohjelman indeksoivan objektin aivan kuin luokka itse olisi taulukko.

Mieti seuraavaa esimerkkiä. Sinulla on luetteloruutuluokka, joka pitää toteuttaa niin, että luokan käyttäjä voi lisätä siihen merkkijonoja. Jos tunnet Win32 SDK:n, tiedät, että luetteloruutuun lisätään merkkijono lähettämällä LB_ADDSTRING tai LB_INSERTSTRING-viesti. Kun tätä tapa tuli käyttöön 1980-luvun lopulla, kuvittelimme olevamme oikeita olio-ohjelmoijia. Mehän lähetimme viestejä objektille, aivan kuten nuo hienot olioperusteiset analyysi- ja suunnittelukirjat käskivät tekemään. Kun C++:n ja Object Pascalin tapaiset olioperusteiset kielet alkoivat nopeasti yleistyä, opimme, että objekteja voidaan käyttää fiksumpien ohjelmointirajapintojen tekemiseen sellaisia tehtäviä varten. Käyttämällä C++:aa ja MFC:tä (Microsoft Foundation Classes), saimme koko joukon luokkia, joiden avulla pystyimme käsittelemään ikkunoita (kuten luetteloruutuja) objekteina niin, että ne tarjosivat jäsenfunktiot, jotka oikeastaan olivat vain kevyt kääreluokka, viestien lähettämiseen ja vastaanottamiseen taustalla olevalta Microsoft Windows -kontrollilta. *ClistBox*-luokan kohdalla (joka on MFC-kääre Windowsin list box -kontrollille) meille annettiin *AddString* ja *InsertString*-jäsenfunktiot niitä tehtäviä varten, jotka aiemmin tehtiin lähettämällä LB_ADDSTRING tai LB_INSERTSTRING-viesti.

Voidakseen kehittää parhaan ja hienoimman kielen, C#:n suunnitteluryhmä mietti edellistä ja ajatteli, "Miksi me emme voi käsitellä objektia, joka sisimmältään on taulukko, kuin taulukkoa?" Kun mietit luetteloruutua, niin eikö se olekin vain taulukollinen merkkijonoja, johon on lisätty näyttö- ja lajitteluominaisuudet? Tästä ajatuksesta syntyi indeksoija-käsite.

Indeksoijan määrittely

Koska ominaisuuksia joskus sanotaan älykkäiksi kentiksi ja indeksoijia älykkäiksi taulukoiksi, tuntuu järkevältä, että ominaisuuksilla ja indeksoijilla on sama syntaksi. Todellakin, indeksoijan määrittely on melkein sama kuin ominaisuuden määrittely kahdella merkittävällä poikkeuksella: Ensiksikin, indeksoija saa *index*-parametrin. Toiseksi, koska luokkaa itseään käytetään taulukon tavoin, *this*-avainsanaa käytetään indeksoijan nimenä. Näet kohta täydellisemmän esimerkin, mutta katsotaan ensin lyhyttä indeksoija-esimerkkiä:

```
class MyClass
{
    public object this [int idx]
    {
        get
        {
            // Palautetaan haluttu tieto.
        }
        set
        {
            // Asetetaan haluttu tieto.
        }
    }
    ...
}
```

En esittänyt kokonaista toimivaa esimerkkiä indeksoijan toiminnasta, koska tietojesi sisäinen esitysmuoto ja se, miten käsittelet niitä, ei ole indeksoijan kannalta oleellista. Pidä mielessäsi, että riippumatta siitä, miten sisäisesti tallennat tietosi (eli ovatko ne taulukossa, kokoelmassa tai jossain muussa rakenteessa), indeksoijat tarkoittavat yksinkertaisesti , että ohjelmoija voi instantioida luokan seuraavanlaisella koodilla:

```
MyClass cls = new MyClass();
cls[0] = someObject;
Console.WriteLine("{0}", cls[0]);
```

Se, mitä teet indeksoijilla, on oma asiasi, kunhan luokan käyttäjät saavat odottamansa tulokset käsitellessään objektia taulukkona.

Indeksoijaesimerkki

Katsotaanpa tilanteita, joissa indeksoijista on eniten hyötyä. Aloitan jo aiemmin käyttämälläni luetteloruutuesimerkillä. Kuten todettua, käsitteellisessä mielessä luetteloruutu on yksinkertainen näytettävä luettelo tai taulukko merkkijonoja. Seuraavassa esimerkissä määrittelen luokan nimeltä *MyListBox*, joka sisältää indeksoijan merkkijonojen lukemiseen ja asettamiseen *ArrayList*-objektin kautta. (*ArrayList* on .NET Framework -luokka, jota käytetään objektikokoelman säilyttämiseen.)

```
using System;
using System.Collections;

class MyListBox
{
    protected ArrayList data = new ArrayList();
```

```
    public object this[int idx]
    {
        get
        {
            if (idx > -1 && idx < data.Count)
            {
                return (data[idx]);
            }
            else
            {
                // Mahdollisesti aiheutetaan tässä poikkeus.
                return null;
            }
        }
        set
        {
            if (idx > -1 && idx < data.Count)
            {
                data[idx] = value;
            }
            else if (idx == data.Count)
            {
                data.Add(value);
            }
            else
            {
                // Mahdollisesti aiheutetaan tässä poikkeus.
            }
        }
    }
}

class Indexers1App
{
    public static void Main()
    {
        MyListBox lbx = new MyListBox();
        lbx[0] = “foo";
        lbx[1] = “bar";
        lbx[2] = “baz";
        Console.WriteLine(“{0} {1} {2}",
                          lbx[0], lbx[1], lbx[2]);
    }
}
```

Huomaa, että tässä esimerkissä tarkistan out-of-bounds-virheen tietojen indeksoinnissa. Tämä ei ole teknisesti sidottu indeksoijiin, sillä kuten mainitsin, indeksoijiin liittyy ainoastaan se, miten luokan käyttäjät voivat käyttää objekteja taulukon tavoin eikä

sillä ole mitään tekemistä tietojen sisäisen esitystavan kanssa. Kun opettelee uutta kieltä, ominaisuutta hyödyntävän käytännöllisen esimerkin näkeminen auttaa enemmän kuin pelkkä ominaisuuden syntaksi. Siis, sekä indeksoijan getter- että setter-metodissa tarkistan välitetyn indeksiarvon kelpoisuuden luokan *ArrayList*-jäseneen tallennettuja tietoja vasten. Henkilökohtaisesti luultavasti valitsisin poikkeuksen aiheuttamisen tapauksissa, joissa välitetyn parametrin arvo ei ole kelvollinen. Se on kuitenkin henkilökohtainen päätös, saatat itse käsitellä virhetilanteet eri tavalla. Oleellista on, että ilmaiset virhetilanteen käyttäjälle, jos välitetyn parametrin arvo ei ole kelvollinen.

Suunnitteluohjeita

Indeksoijat ovat taas yksi esimerkki C#-tiimin kieleen lisäämistä pienistä mutta tehokkaista ominaisuuksista, jotka auttavat meitä tehokkaimmiksi ohjelmoijiksi. Kuten jokaisella muullakin kielen ominaisuudella, on indeksoijalla paikkansa. Niitä tulee käyttää vain paikoissa, joissa on luonnollista käsitellä objekteja taulukon tavoin. Ajatellaan laskutussovellusta. On järkevää, että sovelluksella on *Invoice*-luokka, joka sisältää jäsentaulukon *InvoiceDetail*-objekteja. Tällöin on käyttäjän kannalta täysin luonnollista käsitellä näitä laskurivejä seuraavasti:

```
InvoiceDetail detail = invoice[2]; // Haetaan 3. laskurivi.
```

Sen sijaan ei olisi järkevää mennä askel pidemmälle ja yrittää kääntää kaikki *InvoiceDetail*-luokan jäsenet taulukoiksi, joita käsiteltäisiin indeksoijan avulla. Kuten näet seuraavasta, ensimmäinen rivi on paljon ymmärrettävämpi kuin toinen:

```
TermCode terms = invoice.Terms;  // Ominaisuuskäsittelijä Terms-
jäseneen.
TermCode terms = invoice[3];     // Huono ratkaisu olemattomaan
ongelmaan.
```

Tässä(kin) tapauksessa kannattaa pitää mielessä, että älä tee mitään pelkästään sen takia, että voit tehdä sitä. Tai konkreettisemmin sanottuna, mieti miten jonkin uuden ominaisuuden toteuttaminen vaikuttaa luokan käyttäjiin ja anna sen ajatuksen johdattaa sinua, kun päätät, toteutatko ominaisuuden, joka tekee luokkasi käyttämisen helpommaksi.

Yhteenveto

C#:n ominaisuudet koostuvat kentän määrittelystä ja käsittelymetodeista. Ominaisuuksien avulla voit käsitellä luokan kenttiä älykkäästi eikä luokkaa käyttävän ohjelmoijan tarvitse tutkia, onko (ja miten) kentän käsittelymetodit toteutettu. Taulukot määritellään C#:ssa sijoittamalla tyhjät hakasulkeet tyypin ja muuttujan nimen väliin eli tapa eroaa hieman C++:n taulukon määrittelystä. C#-taulukot voivat olla yksiulotteisia, moniulotteisia tai sisäkkäisiä. Objektia voidaan indeksoijan avulla C#:ssa käsitellä kuin taulukoita. Indeksoijan avulla ohjelmoija voi helposti käsitellä ja tutkia useita saman tyyppisiä objekteja.