9

Rajapinnat

Rajapinnan ymmärtämisen avain saattaa olla sen vertaaminen luokkaan. Luokat ovat objekteja, joilla on ominaisuuksia ja metodeja, jotka käyttävät noita ominaisuuksia. Vaikka luokat sisältävät käyttäytymisominaisuuksia (metodeja), luokat ovat asioita vastakohtana käyttäytymiselle ja siinä tulevat rajapinnat esiin. Rajapinnan avulla voit määritellä käyttäytymisominaisuudet, tai kyvyt, ja soveltaa niitä luokkiin luokkahierarkkiasta riippumatta. Sanotaan esimerkiksi, että sinulla on hajautettu sovellus, jossa jotkin käsitteet voidaan serialisoida (tallentaa). Tällaisia ovat vaikkapa luokat *Customer*, *Supplier* ja *Invoice*. Jotkut muut luokat taas, kuten vaikkapa *MaintenanceView* ja *Document*, eivät ole serialisoitavia. Miten teet vain haluamistasi luokista serialisoitavia? Yksi selvä tapa olisi tehdä kantaluokka nimeltä *Serializable*. Tällä ratkaisulla on kuitenkin suuria puutteita. Yhden periytymisen polku ei toimi, koska emme halua kaikkien luokan ominaisuuksien olevan jaettuja. C# ei tue moniperintää, joten ei ole mahdollista periyttää luokka valinnan mukaan useista luokista. Vastaus on rajapinnat. Niiden avulla voit määritellä joukon toisiinsa liittyviä metodeja ja ominaisuuksia, jotka määrätyt luokat voivat toteuttaa luokkahierarkkiasta riippumatta.

Käsitteelliseltä kannalta rajapinnat ovat kahden eri koodin välinen sopimus. Siis, kun rajapinta on määritelty ja luokka määritelty toteuttamaan tuon rajapinnan, luokan käyttäjät voivat olla varmoja, että luokka on toteuttanut kaikki rajapinnassa määritellyt metodit. Näet tästä kohta muutaman esimerkin.

Tässä luvussa näemme, miksi rajapinnat ovat niin tärkeässä roolissa C#:ssa ja yleensäkin komponenttiperusteisessa ohjelmoinnissa. Sitten katsomme miten määritellään ja toteutetaan rajapinta C#-sovelluksessa. Lopuksi tutkimme tarkemmin määrityksiä, joita rajapintojen käyttöön liittyy ja selvittelemme moniperinnän ja nimien sekaantumisen ongelmia.

Huomaa Kun määrittelet rajapinnan ja ilmoitat, että luokka tulee käyttämään sitä määrittelyssään, sanotaan, että *luokka toteuttaa rajapinnan* tai *luokka periytyy rajapinnasta*. Voit käyttää kumpaakin ilmausta ja tulet huomaamaan, että niitä käytetään vaihtelevasti muissa teksteissä. Henkilökohtaisesti uskon, että toteuttaa on semanttisesti oikeampi termi. Rajapinnat ovat määriteltyjä käyttäytymisiä ja luokat on määritelty toteuttamaan tuon käyttäytyminen vastakohtana periytymiselle toisesta luokasta. Mutta molemmat ilmaukset ovat oikein.

Rajapinnan käyttäminen

Ymmärtääksesi, missä rajapinnat ovat hyödyllisiä, katsotaan ensin perinteistä ohjelmointiongelmaa Microsoft Windows -ympäristössä, joka ei käytä rajapintaa, mutta jossa kaksi erillistä koodia pitäisi saada keskustelemaan keskenään yleispätevällä tavalla. Kuvitellaan, että työskentelet Microsoftilla ja olet pääsuunnittelijana Control Panel -ryhmässä (Ohjauspaneeli). Sinun tulee tarjota yleinen menetelmä, jolla mikä tahansa asiakasappletti voidaan ”pultata” Control Paneliin niin, että sen kuvake ilmestyy Control Panel -ikkunaan ja käyttäjä voi käynnistää sen siitä. Pitäen mielessä, että tämä toiminnallisuus on kehitetty ennen kuin COM esiteltiin, niin miten järjestäisit minkä tahansa tulevan sovelluksen integroitumisen Control Paneliin? Aikanaan kehitetty ratkaisu on ollut vuosia oleellinen osa Windows-ohjelmointia.

Control Panel -ryhmän pääsuunnittelijana suunnittelet ja dokumentoit funktion (tai funktiot), jotka asiakassovelluksen tulee toteuttaa määrättyjen sääntöjen mukaan. Control Paneelin applettien tapauksessa, Microsoft määritteli, että kirjoittaaksesi sellaisen, sinun pitää tehdä dll (dynamic-link library), joka toteuttaa funktion nimeltä CPIApplet. Sinun tulee myös lisätä tämän dll:n nimeen tarkenne .cpl ja sijoittaa se Windowsin System32-kansioon. (Windows ME:ssä ja Windows 98:ssa kansion nimi on Windows\System32 ja Windows 2000:ssa se on WINNT\System32.) Kun Control Panel latautuu, se lataa kaikki System32-kansiossa olevat .cpl-päätteiset dll:t (käyttämällä LoadLibrary-funktiota) ja sen jälkeen käyttää GetProcAddress-funktiota ladatakseen CPIApplet-funktion, siten tarkistaen, että olet noudattanut ohjeita ja dll:si siirtää tietoja oikein Control Panelin kanssa.

Kuten mainitsin, tämä on standardi ohjelmointimalli Windowsissa sellaisten tilanteiden käsittelyyn, joissa sinulla on koodinpalanen, jonka haluat kommunikoivan tulevaisuuden tuntemattomien koodien kanssa yleispätevällä tavalla. Tämä ei kuitenkaan ole maailman fiksuin ratkaisu ja sillä on ilmiselvästi omat rajoituksensa. Huonoin puoli tässä tekniikassa on se, että se pakottaa asiakasohjelman, tässä tapauksessa Control Panel -koodin, sisällyttävän itseensä paljon erilaisia tarkistuksia. Control Panel ei voi esimerkiksi olettaa, että jokainen kansiossa oleva .cpl-päätteinen tiedosto on Windows dll. Sen pitää

myöskin tarkistaa, että dll sisältää korjausfunktiot ja että funktiot tekevät mitä dokumentti vaatii. Tässä tulevat rajapinnat esille. Rajapintojen avulla voit luoda saman sopimusperusteisen järjestelyn kahden koodin välille mutta olioperusteisemmalla ja joustavammalla tavalla. Lisäksi, koska rajapinnat ovat osa C#-kieltä, kääntäjä varmistaa, että kun luokka on määritelty toteuttamaan annettu rajapinta, se myös tekee sen minkä lupaa.

C#:ssa rajapinta on ensisijainen menetelmä, joka määrittelee viittaustyypin, joka sisältää vain metodin määrittelyn. Mutta mitä tarkoitan, kun sanon "ensisijainen?" Tarkoitan, että kyseessä oleva ominaisuus on sisään rakennettu, oleellinen osa kieltä. Toisin sanoen, se ei ole mitään kielen suunnittelun jälkeen lisättyä. Sukelletaan nyt rajapintojen yksityiskohtiin ja niiden määrittelyyn.

> Huomaa C++-ohjelmoijia ajatellen: rajapinta on pohjimmiltaan abstrakti luokka, jossa on määritelty vain puhtaita virtuaalisia metodeja muiden C#-luokkien jäsenten, kuten ominaisuuksien, tapahtumien ja indeksoijien lisäksi.

Rajapintojen määritteleminen

Rajapinnat voivat sisältää metodeja, ominaisuuksia, indeksoijia ja tapahtumia, joista rajapinta ei itse toteuta yhtäkään. Katsotaan esimerkkiä, jotta näemme, miten tätä piirrettä käytetään. Oletetaan, että olet suunnittelemassa yrityksellesi editoria, joka sisältää erilaisia Windows-kontrolleja. Kirjoitat editorin ja rutiinit, jotka tarkistavat käyttäjien lomakkeelle sijoittamat kontrollit. Ryhmän muu osa kirjoittaa kontrolleja, joita lomakkeelle sijoitetaan. Melko varmasti sinun pitää toteuttaa jonkinlainen lomaketason tarkistus. Sopivin väliajoin, esimerkiksi kun käyttäjä pyytää sitä tai lomakkeen generoinnin aikana, lomake voi käydä läpi kaikki itseensä sijoitetut kontrollit ja tutkia kunkin kelpoisuus tai täsmällisemmin sanottuna, pyytää kutakin kontrollia tarkistamaan itsensä.

Miten tarjoat tämän kontrollin kelpoisuustarkistuksen? Tällaisessa tilanteessa rajapinta on erityisen sopiva. Tässä yksinkertainen esimerkki rajapinnasta, joka sisältää yhden *Validate*-nimisen metodin.

```
interface IValidate
{
    bool Validate();
}
```

Nyt voit varmistaa ohjelmallisesti, että jos kontrolli toteuttaa *IValidate*-rajapinnan, sen kelpoisuus voidaan tutkia.

Tutkitaanpa muutamaa näkökohtaa edellisessä koodipätkässä. Ensinnäkään sinun ei tarvitse määritellä käsittelymääreitä (esimerkiksi *public*) rajapinnan metodeille. Itse asiassa sellaisen kirjoittaminen metodin nimen eteen aiheuttaisi kääntäjän virheilmoituksen. Tämä johtuu siitä, että kaikki rajapinnan metodit ovat oletuksena julkisia (*public*). (C++-ohjelmoijat voivat myös havaita, että koska rajapinnat määrittelyn mukaan ovat abstrakteja luokkia, niitä ei tarvitse erikseen määritellä puhtaaksi virtuaalifunktioksi lisäämällä =0 metodin määritykseen.)

Metodien lisäksi rajapinnat voivat määritellä ominaisuuksia, indeksoijia ja tapahtumia, kuten seuraavassa näet:

```
interface IExampleInterface
{
    // Esimerkki ominaisuuden määrittelystä.
    int testProperty { get; }

    // Esimerkki tapahtuman määrittelystä.
    event testEvent Changed;

    // Esimerkki indeksoijan määrittelystä.
    string this[int index] { get; set; }
}
```

Rajapintojen toteuttaminen

Koska rajapinta määrittelee sopimuksen, jokaisen sen toteuttavan luokan *tulee toteuttaa jokainen rajapinnan osa* tai koodi ei käänny. Jos käytät edellisen esimerkin *IValidate*-rajapintaa, tulee luokasi toteuttaa vain sen metodit. Seuraavassa esimerkissä minulla on kantaluokka *FancyControl* ja rajapinta *IValidate*. Minulla on myös luokka *MyControl*, joka periytyy luokasta *FancyControl* ja joka toteuttaa rajapinnan *IValidate*. Huomaa syntaksi ja se, miten *MyControl*-objekti voidaan muuntaa (cast) *IValidate*-rajapinnaksi sen jäseniin viittaamista varten.

```
using System;

public class FancyControl
{
    protected string Data;
        public string data
        {
            get
```

```
        {
            return this.Data;
        }
        set
        {
            this.Data = value;
        }
    }
}

interface IValidate
{
    bool Validate();
}

class MyControl : FancyControl, IValidate
{
    public MyControl()
    {
        data = “my grid data";
    }

    public bool Validate()
    {
        Console.WriteLine(“Validating...{0}", data);
                return true;
    }
}

class InterfaceApp
{
    public static void Main()
    {
        MyControl myControl = new MyControl();

        // Kutsutaan funktiota, joka sijoittaa
        // kontrollin lomakkeelle. Nyt, kun editorin
        // pitää tarkistaa kontrolli, se voi
        // tehdä sen seuraavasti:

        IValidate val = (IValidate)myControl;
        bool success = val.Validate();
    Console.WriteLine(“The validation of ‘{0}’ was {1}successful",
                      myControl.data,
                       (true == success ? “” : “not “));
    }
}
```

Käyttämällä edellä olevaa luokan ja rajapinnan määrittelyä, editori voi käydä kontrollit läpi tutkien, toteuttaako se *IValidate*-rajapinnan. (Näet seuraavassa kappaleessa miten se tehdään.) Jos toteuttaa, editori voi kutsua toteutettua rajapinnan metodia. Voit tässä vaiheessa ehkä kysäistä, "Miksi en määrittele kantaluokkaa tälle editorille, jossa olisi puhdas virtuaalinen funktio *Validate*? Editori voi sitten tutkia vain ne kontrollit, jotka periytyvät tästä kantaluokasta, eikö?"

Tämä olisi toki toimiva ratkaisu, mutta sillä on muutamia rajoituksia. Sanotaan, että teet omia kontrolleja ja ne periytyisivät tästä kuvitteellisesta luokasta. Täten ne kaikki toteuttavat tämän virtuaalisen *Validate*-metodin. Tämä toimii, kunnes jonakin päivänä löydät todella fiksun kontrollin, jota haluat käyttää editorissasi. Sanotaan, että se on jonkun muun tekemä taulukko-kontrolli. Siten se ei periydy editorisi pakollisesta kantaluokasta. C++:ssa vastaus olisi moniperintä eli periyttäisit oman taulukkosi sekä tästä ulkopuolisesta taulukosta että editorin kantaluokasta. C# ei kuitenkaan tue moniperintää.

Käyttämällä rajapintoja voit toteuttaa useita toiminnallisia ominaisuuksia yhdessä luokassa. C#:ssa voit periyttää yhdestä luokasta ja sen toiminnallisuuden lisäksi toteuttaa niin monta rajapintaa kuin luokka tarvitsee. Jos esimerkiksi haluaisit editori-sovelluksen tarkistavan kontrollin sisällön, liittää kontrollin tietokantaan ja serialisoida sen sisällön levylle, voit määritellä luokkasi seuraavasti:

```
public class MyGrid : ThirdPartyGrid, IValidate,
                      ISerializable, IDataBound
{
§
}
```

Kuten lupasin, seuraava kappale vastaa kysymykseen, "Miten koodilla voi selvittää, milloin luokka toteuttaa annetun rajapinnan?"

Toteutuksen kyseleminen *is*-operaattorilla

InterfaceApp-esimerkissä näit seuraavan koodin, jota käytettiin muuntamaan objekti (*MyControl*) yhdeksi sen toteuttamista rajapinnoista (*IValidate*) ja sen jälkeen kutsumaan tuon rajapinnan jäsentä (*Validate*):

```
MyControl myControl = new MyControl();
§
IValidate val = (IValidate)myControl;
bool success = val.Validate();
```

Mitä tapahtuisi, jos asiakasohjelma yrittäisi käyttää luokkaa aivan kuin se olisi toteuttanut metodin, jota ei olekaan? Seuraava esimerkki kääntyy, koska *ISerializable* on

kelvollinen rajapinta. Silti suorituksen aikana aiheutuu *System.InvalidCastException* poikkeus, koska *MyGrid* ei toteuta *ISerializable*-rajapintaa. Sovelluksen suoritus keskeytyisi, jos poikkeusta ei käsitellä asiallisesti.

```
using System;

public class FancyControl
{
    protected string Data;
    public string data
    {
        get
        {
            return this.Data;
        }
        set
        {
            this.Data = value;
        }
    }
}

interface ISerializable
{
    bool Save();
}

interface IValidate
{
    bool Validate();
}

class MyControl : FancyControl, IValidate
{
    public MyControl()
    {
        data = “my grid data";
    }

    public bool Validate()
    {
        Console.WriteLine(“Validating...{0}", data);
        return true;
    }
}
```

(jatkuu)

```
class IsOperator1App
{
    public static void Main()
    {
        MyControl myControl = new MyControl();

        ISerializable ser = (ISerializable)myControl;

        // HUOMAA: Tästä tulee System.InvalidCastException-
        // poikkeus, koska luokka ei toteuta
        // ISerializable-rajapintaa.
        bool success = ser.Save();

        Console.WriteLine("The saving of '{0}' was {1}successful",
 myControl.data,
 (true == success ? "" : "not "));
    }
}
```

Poikkeuksen kiinni ottaminen ei tietenkään muuta sitä tosiasiaa, että koodi ei tässä tapauksessa toimi. Sinun pitää jollakin tavalla selvittää objektin olemus *ennen* kuin teet tyyppimuunnoksen. Yksi tapa on tehdä se *is*-operaattorilla. Sen avulla voit tarkistaa ohjelman suorituksen aikana, onko määrätty tyyppi yhteensopiva toisen kanssa. Sitä käytetään seuraavasti (*expression* on tyyppi, johon viittaus on):

expression is *type*

Is-operaattori palauttaa Boolean-arvon ja siksi sitä voidaan käyttää ehtolauseissa. Seuraavassa olen muokannut koodia ja testaan yhteensopivuutta *MyControl*-luokan ja *ISerializable*-rajapinnan välillä ennen kuin yritän käyttää *ISerializable*-rajapinnan metodia:

```
using System;

public class FancyControl
{
    protected string Data;
    public string data
    {
        get
        {
            return this.Data;
        }
        set
        {
            this.Data = value;
        }
```

```
    }
}

interface ISerializable
{
    bool Save();
}

interface IValidate
{
    bool Validate();
}

class MyControl : FancyControl, IValidate
{
    public MyControl()
    {
        data = “my grid data";
    }

    public bool Validate()
    {
        Console.WriteLine(“Validating...{0}", data);
        return true;
    }
}

class IsOperator2App
{
    public static void Main()
    {
        MyControl myControl = new MyControl();

        if (myControl is ISerializable)
        {
            ISerializable ser = (ISerializable)myControl;
            bool success = ser.Save();

            Console.WriteLine(“The saving of ‘{0}’ was {1}successful",
myControl.data,
(true == success ? “” : “not “));
        }
        else
        {
         Console.WriteLine(“The ISerializable interface is not implemented.”);
        }
    }
}
```

Nyt, kun olet nähnyt, miten *is*-operaattorin avulla voit varmistaa kahden tyypin yhteensopivuuden oikean käytön varmistamiseksi, katsotaan yhtä sen läheistä sukulaista eli *as*-operaattoria ja verrataan näitä kahta.

Toteutuksen kyseleminen *as*-operaattorilla

Jos katsot tarkkaan edellisen IsOperator2App-sovelluksesta generoitua MSIL-koodia (koodi on nähtävissä tämän kappaleen jälkeen), huomaat yhden ongelman *is*-operaattorissa. Näet, että *isinst*-käskyä kutsutaan heti, kun objektin tilavaraus on tehty ja pino kunnossa. *Isinst*-käskyn kääntäjä on generoinut C#:n *is*-operaattorista. Käsky tutkii, onko objekti luokan tai rajapinnan instanssi. Huomaa, että vain muutamaa riviä myöhemmin (olettaen, että ehtolause toteutuu), kääntäjä on generoinut *castclass*-käskyn. *Castclass*-käsky tekee oman tarkistuksensa ja vaikka tämä käsky toimii hieman eri tavalla kuin *isinst*, tuloksena on, että generoitu IL tekee tehotonta työtä tarkistamalla tyyppimuunnoksen kelvollisuuden kahdesti.

```
.method public hidebysig static void Main() il managed
{
  .entrypoint
  // Code size       72 (0x48)
  .maxstack  4
  .locals (class MyControl V_0,
           class ISerializable V_1,
           bool V_2)
  IL_0000:  newobj     instance void MyControl::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
  IL_0007:  isinst     ISerializable
  IL_000c:  brfalse.s  IL_003d
  IL_000e:  ldloc.0
  IL_000f:  castclass  ISerializable
  IL_0014:  stloc.1
  IL_0015:  ldloc.1
  IL_0016:  callvirt   instance bool ISerializable::Save()
  IL_001b:  stloc.2
  IL_001c:  ldstr      “The saving of ‘{0}’ was {1}successful"
  IL_0021:  ldloc.0
  IL_0022:  call       instance class System.String FancyControl::get_data()
  IL_0027:  ldloc.2
  IL_0028:  brtrue.s   IL_0031
  IL_002a:  ldstr      “not “
  IL_002f:  br.s       IL_0036
  IL_0031:  ldstr      “"
  IL_0036:  call   void [mscorlib]System.Console::WriteLine(class System.String,
                                                             class System.Object,
```

```
                                                          class System.Object)
  IL_003b:  br.s   IL_0047
  IL_003d:  ldstr  "The ISerializable interface is not implemented."
  IL_0042:  call  void [mscorlib]System.Console::WriteLine(class System.String)
  IL_0047:  ret
} // end of method IsOperator2App::Main
```

Voimme tehdä tästä tarkistusprosessista tehokkaamman käyttämällä *as*-operaattoria. *as*-operaattori tekee kahden yhteensopivan tyypin välisen tyyppimuunnoksen. Sitä käytetään seuraavasti (expression on tyyppi, johon viittaus on):

object = *expression* as *type*

Voit ajatella *as*-operaattoria *is*-operaattorin ja (jos tyypit ovet yhteensopivia) tyyppimuunnoksen yhdistelmänä. Merkittävä ero *as*-operaattorin ja *is*-operaattorin välillä on se, että *as* asettaa objektin arvoksi *null*, jos *expression* ja *type* eivät ole yhteensopivia, sen sijaan, että palauttaisi Boolean-arvon. Esimerkkimme voidaan nyt kirjoittaa tehokkaammalla tavalla seuraavasti:

```
using System;

public class FancyControl
{
    protected string Data;
    public string data
    {
        get
        {
            return this.Data;
        }
        set
        {
            this.Data = value;
        }
    }
}

interface ISerializable
{
    bool Save();
}

interface IValidate
{
    bool Validate();
```

(jatkuu)

```
}

class MyControl : FancyControl, IValidate
{
    public MyControl()
    {
        data = “my grid data";
    }

    public bool Validate()
    {
        Console.WriteLine(“Validating...{0}", data);
        return true;
    }
}

class AsOperatorApp
{
    public static void Main()
    {
        MyControl myControl = new MyControl();

        ISerializable ser = myControl as ISerializable;
        if (null != ser)
        {
            bool success = ser.Save();

            Console.WriteLine(“The saving of ‘{0}’ was {1}successful",
myControl.data,
(true == success ? “” : “not “));
        }
        else
        {
          Console.WriteLine(“The ISerializable interface is not implemented.”);
        }
    }
}
```

Nyt tarkistus tyyppimuunnosta varten tehdään vain kerran, ja se on tietenkin selvästi tehokkaampaa. Palataan tässä vaiheessa tutkimaan MSIL-koodia ja katsotaan, mitä vaikutuksia *as*-operaattorin käytöstä oli:

```
 .method public hidebysig static void Main() il managed
{
  .entrypoint
  // Code size       67 (0x43)
```

```
  .maxstack  4
  .locals (class MyControl V_0,
           class ISerializable V_1,
           bool V_2)
  IL_0000:  newobj     instance void MyControl::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
  IL_0007:  isinst     ISerializable
  IL_000c:  stloc.1
  IL_000d:  ldloc.1
  IL_000e:  brfalse.s  IL_0038
  IL_0010:  ldloc.1
  IL_0011:  callvirt   instance bool ISerializable::Save()
  IL_0016:  stloc.2
  IL_0017:  ldstr      “The saving of ‘{0}’ was {1}successful"
  IL_001c:  ldloc.0
  IL_001d:  call       instance class System.String FancyControl::get_data()
  IL_0022:  ldloc.2
  IL_0023:  brtrue.s   IL_002c
  IL_0025:  ldstr      “not “
  IL_002a:  br.s       IL_0031
  IL_002c:  ldstr      “"
  IL_0031:  call   void [mscorlib]System.Console::WriteLine(class System.String,
                                                             class System.Object,
                                                             class System.Object)
  IL_0036:  br.s       IL_0042
  IL_0038:  ldstr      “The ISerializable interface is not implemented."
  IL_003d:  call   void [mscorlib]System.Console::WriteLine(class System.String)
  IL_0042:  ret
} // end of method AsOperatorApp::Main
```

Explisiittinen rajapinnan jäsenen nimen määrittäminen

Tähän mennessä olet nähnyt luokkien toteuttavan rajapintojen jäseniä määrittelemällä käsittelymääreen *public*, jota on seurannut rajapinnan metodin kutsu parametreineen. Joskus kuitenkin haluat (tai sinun jopa pitää) explisiittisesti määritellä jäsenen nimi ja rajapinnan nimi. Tässä kappaleessa käsittelemme kaksi tapausta, jotka aiheuttavat tällaisen tarpeen.

Nimen piilottaminen rajapinnalla

Yleisin tapa kutsua rajapinnasta toteutettua metodia on tehdä luokan instanssille tyyppimuunnos (cast) rajapintatyypiksi ja sen jälkeen kutsua haluttua metodia. Vaikka tämä on kelvollinen tapa ja monet (mukaan lukien minä) käyttävät tätä tekniikkaa, sinun ei teknisesti tarvitse tehdä tyyppimuunnosta toteutettuun rajapintatyyppiin, jotta voisit kutsua

sen metodeja. Syy on se, että koska luokka toteuttaa rajapinnan metodit, nuo metodit ovat myös luokan julkisia metodeja. Katsotaan seuraavaa C#-koodia ja erityisesti sen *Main*-metodia, niin näet, mitä tarkoitan:

```
using System;

public interface IDataBound
{
    void Bind();
}

public class EditBox : IDataBound
{
    // IDataBound-rajapinnan toteutus.
    public void Bind()
    {
        Console.WriteLine("Binding to data store...");
    }
}

class NameHiding1App
{
    // Ohjelman alkukohta.
    public static void Main()
    {
        Console.WriteLine();

        EditBox edit = new EditBox();
        Console.WriteLine("Calling EditBox.Bind()...");
        edit.Bind();

        Console.WriteLine();

        IDataBound bound = (IDataBound)edit;
        Console.WriteLine("Calling (IDataBound)EditBox.Bind()...");
        bound.Bind();
    }
}
```

Tämä esimerkki aiheuttaa seuraavat tulokset:

```
Calling EditBox.Bind()...
Binding to data store...

Calling (IDataBound))EditBox.Bind()...
Binding to data store...
```

Huomaa, että vaikka tämä sovellus kutsui toteutettua *Bind*-metodia kahdella erilaisella tavalla, tyyppimuunnoksella ja ilman, molempien funktiokutsujen tuloksena kutsuttiin metodia oikein. Vaikka ensimmäinen osoitti mahdollisuutta kutsua toteutettua metodia suoraan ilman tyyppimuunnosta rajapintatyypiksi, tämä ei aina ole kuitenkaan toivottava mahdollisuus. Selvin syy on se, että useiden rajapintojen (joista kukin voi sisältää useita jäseniä) toteuttaminen täyttää nopeasti luokkasi yleisen nimiavaruuden jäsenillä, joilla ei ole mitään merkitystä toteutetun luokan näkyvyysalueen ulkopuolella. Voit estää rajapintojen toteutettujen jäsenten tulemisen luokan julkisiksi jäseniksi käyttämällä *nimen piilottamista* (name hiding).

Nimen piilottaminen on yksinkertaisimmillaan mahdollisuus piilottaa periytetyn jäsenen nimi kaikelta periytetyn tai toteutetun luokan ulkopuoliselta (josta käytetään usein nimitystä ulkopuolinen maailma). Otetaan taas käyttöön se vanha esimerkki, jossa *EditBox*-luokan piti toteuttaa *IDataBound*-rajapinta. Tällä kertaa *EditBox*-luokka ei halua näyttää *IDataBound*-rajapinnan metodeja ulkopuoliselle maailmalle, vaan tarvitsee rajapintaa omiin tarkoituksiinsa tai ehkä ohjelmoija ei yksinkertaisesti halua sekoittaa luokan nimiavaruutta suurella määrällä metodeja, joita tavallinen asiakasohjelma ei tarvitse. Piilottaaksesi toteutetun rajapinnan jäsenen sinun pitää vain poistaa jäsenen käsittelymääre *public* ja ilmoittaa jäsenen nimi yhdessä sen rajapinnan nimen kanssa, näin:

```
using System;

public interface IDataBound
{
    void Bind();
}

public class EditBox : IDataBound
{
    // IDataBound implementation.
    void IDataBound.Bind()
    {
        Console.WriteLine("Binding to data store...");
    }
}

class NameHiding2App
{
    public static void Main()
    {
        Console.WriteLine();
```

(jatkuu)

```
            EditBox edit = new EditBox();
            Console.WriteLine("Calling EditBox.Bind()...");

            // VIRHE: Tämä rivi ei käänny, koska
            // Bind-metodia ei ole enää olemassa
            // EditBox-luokan nimiavaruudessa.
            edit.Bind();

            Console.WriteLine();

            IDataBound bound = (IDataBound)edit;
            Console.WriteLine("Calling (IDataBound)EditBox.Bind()...");

            // Tämä on OK, koska objekti on ensin
            // tyyppimuunnettu IDataBound-rajapinnaksi
            bound.Bind();
        }
    }
```

Edeltävä koodi ei käänny, koska *Bind*-niminen jäsen ei ole enää osa *EditBox*-luokkaa. Siksi voit tämän menetelmän avulla poistaa jäsenen luokan nimiavaruudesta ja silti yhä sallia sen eksplisiittisen käyttämisen tyyppimuunnoksen avulla.

Haluan vielä toistaa, että kun piilotat jäsenen, et voi käyttää käsittelymäärettä. Saat käännösvirheen, jos yrität antaa käsittelymääreen toteutetulle rajapinnan jäsenelle. Tämä saattaa tuntua oudolta, mutta muista, että perussyy jonkin piilottamiseen on estää sitä näkymästä luokan ulkopuolelle. Koska käsittelymääreiden tarkoitus on ilmoittaa näkyvyystaso kantaluokan ulkopuolelle, ymmärrät, että niiden käytössä ei ole mieltä kun käytät nimen piilotusta.

Nimiristiriitojen välttäminen

Yksi suurimmista syistä, miksi C# ei tue moniperintää, on nimisekaannukset, jotka aiheutuvat nimien ristiriidoista. Vaikka C# ei tue moniperintää objektitasolla (periytymisenä luokasta), se tukee periytymistä yhdestä luokasta ja lisäksi useiden rajapintojen toteuttamista. Tällä teholla on kuitenkin hintansa: nimisekaannukset.

Seuraavassa esimerkissä meillä on kaksi rajapintaa, *ISerializable* ja *IDataStore*, jotka tukevat tietojen lukemista ja kirjoittamista kahdessa muodossa, toinen binaarimuodossa levyllä ja toinen tietokantaan. Ongelma on se, että molemmat sisältävät *SaveData*-nimisen metodin:

```
using System;

interface ISerializable
{
    void SaveData();
}

interface IDataStore
{
    void SaveData();
}

class Test : ISerializable, IDataStore
{
    public void SaveData()
    {
        Console.WriteLine(“Test.SaveData called”);
    }
}

class NameCollisions1App
{
    public static void Main()
    {
        Test test = new Test();

        Console.WriteLine(“Calling Test.SaveData()”);
        test.SaveData();
    }
}
```

Tämän kirjoitushetkellä tämä koodi kääntyy. Minulle on kuitenkin kerrottu, että myöhemmissä C#-kääntäjän versioissa koodi aiheuttaa käännösvirheen toteutetun *SaveData*-metodin ristiriidan takia. Koodin kääntymisestä riippumatta sinulla tulee olemaan ongelmia ohjelman suorituksessa, koska toiminta *SaveData*-metodia kutsuttaessa ei ole selvää. Saatko sen *SaveData*-metodin, joka serialisoi objektin levylle vai sen, joka tallentaa sen tietokantaan?

Katso myös seuraavaa koodia:

```
using System;

interface ISerializable
{
    void SaveData();
}
```

(jatkuu)

```
interface IDataStore
{
    void SaveData();
}

class Test : ISerializable, IDataStore
{
    public void SaveData()
    {
        Console.WriteLine("Test.SaveData called");
    }
}

class NameCollisions2App
{
    public static void Main()
    {
        Test test = new Test();

        if (test is ISerializable)
        {
            Console.WriteLine("ISerializable is implemented");
        }

        if (test is IDataStore)
        {
            Console.WriteLine("IDataStore is implemented");
        }
    }
}
```

Tässä *is*-operaattori onnistuu molempien rajapintojen tarkistuksen kanssa, joka osoittaa, että molemmat rajapinnat on toteutettu, vaikka me tiedämme, että niin ei ole! Jopa kääntäjä antaa tämän esimerkin käännöksestä seuraavat varoitukset:

```
NameCollisions2.cs(27,7): warning CS0183: The given expression is
always of the provided ('ISerializable') type
NameCollisions2.cs(32,7): warning CS0183: The given expression is
always of the provided ('IDataStore') type
```

Ongelma on se, että luokka on toteuttanut *Bind*-metodista joko serialisointiversion tai tietokantaversion (ei molempia). Jos asiakasohjelma tutkii toisen rajapinnan toteuttamista (molemmat onnistuvat) ja sattumalta kuitenkin yrittää käyttää sitä versiota, jota ei todellisuudessa ole toteutettu, tapahtuu odottamattomia.

Voit käyttää eksplisiittistä jäsenen nimen määrittelyä päästäksesi eroon tästä ongelmasta: poista käsittelymääre ja lisää jäsenen nimen (tässä tapauksessa *SaveData*) eteen rajapinnan nimi:

```
using System;

interface ISerializable
{
    void SaveData();
}

interface IDataStore
{
    void SaveData();
}

class Test : ISerializable, IDataStore
{
    void ISerializable.SaveData()
    {
        Console.WriteLine(“Test.ISerializable.SaveData called”);
    }
    void IDataStore.SaveData()
    {
        Console.WriteLine(“Test.IDataStore.SaveData called”);
    }
}

class NameCollisions3App
{
    public static void Main()
    {
        Test test = new Test();

        if (test is ISerializable)
        {
            Console.WriteLine(“ISerializable is implemented”);
            ((ISerializable)test).SaveData();
        }

        Console.WriteLine();

        if (test is IDataStore)
        {
            Console.WriteLine(“IDataStore is implemented”);
            ((IDataStore)test).SaveData();
        }
    }
}
```

Nyt ei ole enää epäselvyyttä, mitä metodia kutsutaan. Molemmat metodit on toteutettu täysin määritellyillä nimillään ja sovelluksen suorittaminen antaa odotetut tulokset:

```
ISerializable is implemented
Test.ISerializable.SaveData called

IDataStore is implemented
Test.IDataStore.SaveData called
```

Rajapinnat ja periytyminen

Rajapintoihin ja periytymiseen liittyy kaksi yleistä ongelmaa. Ensimmäinen ongelma, jota seuraava koodi esittää, liittyy tapaukseen, jossa periytetään luokka kantaluokasta, joka sisältää saman nimisen metodin kuin on siinä rajapinnassa, jonka luokan tulee toteuttaa.

```
using System;

public class Control
{
    public void Serialize()
    {
        Console.WriteLine("Control.Serialize called");
    }
}

public interface IDataBound
{
    void Serialize();
}

public class EditBox : Control, IDataBound
{
}

class InterfaceInh1App
{
    public static void Main()
    {
        EditBox edit = new EditBox();
        edit.Serialize();
    }
}
```

Kuten tiedät, toteuttaaksesi rajapinnan sinun tulee tarjota toteutus jokaiselle rajapinnassa määritellylle jäsenelle. Edeltävässä esimerkissä emme kuitenkaan tee sitä ja

silti koodi kääntyy! Syynä on se, että C#-kääntäjä etsii toteutettua *Serialize*-metodia *EditBox*-luokasta ja löytää sellaisen. Kääntäjä kuitenkin virheellisesti määrittelee, että tämä on toteutettu metodi. Kääntäjän löytämä *Serialize*-metodi on *Serialize*-metodi, jonka luokka on perinyt kantaluokasta *Control*, eikä tarkoitettu toteutus *IDataBound.Serialize*-metodista. Vaikka koodi kääntyy, se ei toimi odotusten mukaan, kuten näemme seuraavaksi.

Tehdään nyt asiat vähän mielenkiintoisemmiksi. Huomaa, että seuraava koodi ensin tarkistaa *as*-operaattorilla, että rajapinta on toteutettu ja sen jälkeen yrittää kutsua toteutettua *Serialize*-metodia. Koodi kääntyy ja toimii. Mutta kuten tiedämme, *EditBox*-luokka ei todellisuudessa toteuttanut *Serialize*-metodia *IDataBound*-periytymisen tuloksena. *EditBox*-luokalla jo oli *Serialize*-metodi perittynä *Control*-luokasta. Tämä tarkoittaa, että kaikella todennäköisyydellä asiakasohjelma ei saa odottamiaan tuloksia.

```
using System;

public class Control
{
    public void Serialize()
    {
        Console.WriteLine("Control.Serialize called");
    }
}

public interface IDataBound
{
    void Serialize();
}

public class EditBox : Control, IDataBound
{
}

class InterfaceInh2App
{
    public static void Main()
    {
        EditBox edit = new EditBox();

    IDataBound bound = edit as IDataBound;
    if (bound != null)
    {
        Console.WriteLine("IDataBound is supported...");
        bound.Serialize();
    }
```

(jatkuu)

```
    else
    {
        Console.WriteLine("IDataBound is NOT supported...");
    }
    }
}
```

Toinen tarkkailtava ongelmamahdollisuus on tilanne, jossa periytyvällä luokalla on saman niminen metodi kuin kantaluokan toteutus rajapinnan metodista. Katsotaan myös tätä koodin avulla:

```
using System;

interface ITest
{
    void Foo();
}

// Base toteuttaa rajapinnan ITest.
class Base : ITest
{
    public void Foo()
    {
        Console.WriteLine("Base.Foo (ITest implementation)");
    }
}

class MyDerived : Base
{
    public new void Foo()
    {
        Console.WriteLine("MyDerived.Foo");
    }
}

public class InterfaceInh3App
{
    public static void Main()
    {
        MyDerived myDerived = new MyDerived();
        myDerived.Foo();

        ITest test = (ITest)myDerived;
        test.Foo();
    }
}
```

Tämä koodi saa aikaan seuraavan tulosteen:

```
MyDerived.Foo
Base.Foo (ITest implementation)
```

Tässä tilanteessa *Base*-luokka toteuttaa *ITest*-rajapinnan ja sen *Foo*-metodin. *MyDerived*-luokka perii kuitenkin *Base* -luokan ja toteuttaa uuden *Foo*-metodin. Mitä *Foo*-metodia kutsutaan? Se riippuu viittauksista, joka sinulla on. Jos sinulla on viittaus *MyDerived*-objektiin, kutsutaan sen *Foo*-metodia. Tämä johtuu siitä, että vaikka *myDerived*-objekti on perinyt *ITest.Foo*-metodin toteutuksen, ajonaikainen ympäristö suorittaa *MyDerived.Foo*-metodin, koska new-avainsana määrää perityn metodin korvaamisen.

Kuitenkin, kun explisiittisesti muunnat *myDerived*-objektin *ITest*-rajapinnaksi, kääntäjä ratkaisee kutsut rajapinnan toteutukseen. *MyDerived*-luokalla on saman niminen metodi, mutta kääntäjä ei etsi sitä. Kun teet tyyppimuunnoksen objektista rajapinnaksi, kääntäjä etsii periytymispuusta, kunnes löytyy luokka, joka sisältää rajapinnan. Tämän vuoksi *Main*-metodin kaksi viimeistä riviä aheuttavat *ITets*-rajapinnasta toteutetun *Foo*-metodin kutsumisen.

Toivottavasta nämä nimiristiriitojen ja rajapinnan periytymisen mahdollisesti aiheuttamat ongelmatilanteet saavat tukesi omalle suosituksellenni: tyyppimuunna objekti aina rajapinnaksi, jonka jäsentä aiot käyttää.

Rajapintojen yhdistäminen

Toinen tehokas C#:n ominaisuus on mahdollisuus yhdistää kaksi tai useampia rajapintoja siten, että luokan pitää toteuttaa vain yhdistämisen tulos. Sanotaan esimerkiksi, että haluat luoda uuden *TreeView*-luokan, joka toteuttaa sekä *IDragDrop* että *ISortable*-rajapinnat. Koska on järkevää olettaa, että muutkin kontrollit, kuten *ListView* ja *ListBox*, haluavat myös yhdistää nämä ominaisuudet, kannattaa harkita *IDragDrop* ja *ISortable*-rajapinnan yhdistämistä yhdeksi:

```
using System;

public class Control
{
}

public interface IDragDrop
{
    void Drag();
    void Drop();
}
```

(jatkuu)

```
public interface ISerializable
{
    void Serialize();
}

public interface ICombo : IDragDrop, ISerializable
{
    // Tämä rajapinta ei lisäää mitään uutta
    // käyttäytymistä, vaan sen anoa tarkoitus
    // on yhdistää IDragDrop ja ISerializable
    // rajapinna yhdeksi rajapinnaksi.
}

public class MyTreeView : Control, ICombo
{
    public void Drag()
    {
        Console.WriteLine("MyTreeView.Drag called");
    }

    public void Drop()
    {
        Console.WriteLine("MyTreeView.Drop called");
    }

    public void Serialize()
    {
        Console.WriteLine("MyTreeView.Serialize called");
    }
}

class CombiningApp
{
    public static void Main()
    {
        MyTreeView tree = new MyTreeView();
        tree.Drag();
        tree.Drop();
        tree.Serialize();
    }
```

Rajapintojen yhdistämismahdollisuuden ansiosta et pelkästään helpota semanttisesti toisiinsa liittyvien rajapintojen yhdistämista yhdeksi vaan voit myös lisätä tarpeen mukaan uusia metodeja yhdistettyyn rajapintaan.

Yhteenveto

C#:n rajapintojen avulla voit tehdä luokkia, jotka voivat jakaa ominaisuuksia, mutta jotka eivät ole osa samaa luokkahierarkkiaa. Rajapinnoilla on merkittävä osa C#-ohjelmistokehityksessä, koska C# ei tue moniperintää. Jakaakseen semanttisesti toisiinsa liittyvät metodit ja ominaisuudet, voivat luokat toteuttaa useita rajapintoja. *is* ja *as* -operaattoreita voidaan käyttää määrittämään, onko objekti toteuttanut määrätyn rajapinnan. Tämä joka auttaa estämään rajapinnan jäsenten virheelliseen käyttämiseen liittyviä virhetilanteita. Lopuksi voit käyttää eksplisiittistä jäsenen nimeämistä ja nimen piilottamista ohjatessasi rajapinnan toteuttamista ja estääksesi virhetilanteita.