Metropolia Ammattikorkeakoulu
Tietotekniikan koulutusohjelma

Mikko Vuorinen

LINQ-ohjelmointikieleen yhdistetty hakuarkkitehtuuri

Insinöörityö

Ohjaaja:
Ohjaava opettaja: yliopettaja Kari Aaltonen

Metropolia Ammattikorkeakoulu **Insinöörityön tiivistelmä**

Tekijä Otsikko Sivumäärä Aika	Mikko Vuorinen LINQ-ohjelmointikieleen yhdistetty hakuarkkitehtuuri 100 sivua 10.5.2010
Koulutusohjelma	tietotekniikka
Tutkinto	insinööri (AMK)
Ohjaaja Ohjaava opettaja	 yliopettaja Kari Aaltonen

Insinöörityön aiheena oli esitellä Microsoft .NET Framework 3.5 -ohjelmistokehykseen kuuluva ”Language Integrated Query”-niminen hakuarkkitehtuuri, lyhyemmin LINQ.

LINQ tarjoaa C# 3.0- ja Visual Basic -ohjelmointikieliin täysin integroituvan hakuarkkitehtuurin, jolla voidaan hakea ohjelmallisesti tietoa yhtenäisen hakurajapinnan kautta mistä tahansa tietolähteestä, kuten relaatiotietokannasta, muistissa olevista oliokokoelmista tai XML-tiedostoista.

Insinöörityön tavoitteena oli tuottaa kattava esitys hakuarkkitehtuurin käytöstä ohjelmoinnissa käyttäen esittelyssä hyödyksi ohjelmakoodiesimerkkejä, joita voidaan hyödyntää tekniikan opiskelumateriaalina. Työ on tarkoitettu jo .NET-ohjelmistokehystä tunteville ja C#-ohjelmointikieltä osaaville kokeneemmille ohjelmoijille.

Insinöörityössä esiteltiin C# 3.0 -ohjelmointikielen uudet ominaisuudet, hakuarkkitehtuuri, yleisen hakurajapinnan muodostavat standardihakuoperaattorit ja Linq To SQL -komponentti, joka suorittaa olio-relaatiomallinnuksen relaatiotietokantaa vasten ja hyödyntää hakuarkkitehtuuria tiedon hakemisessa tietokannasta.

Hakusanat	LINQ, Linq To SQL, C# 3.0, .NET Framework 3.5, olio-relaatiomallinnus

Helsinki Metropolia University of Applied Sciences **Abstract**

Author Title Number of Pages Date	Mikko Vuorinen LINQ -language integrated query 100 pages 10 May 2010
Degree Programme	Information Technology
Degree	Bachelor of Engineering
Instructor Supervisor	 Kari Aaltonen, Senior Lecturer

The objective of this bachelor thesis was to present a Microsoft .NET Framework 3.5 feature called Language Integrated Query or LINQ for short.

LINQ provides a fully integrated query architecture for the C# and Visual Basic programming languages, which can be used to query data programmatically through a standardized interface from any kind of data sources, like relational databases, in-memory collections or XML files.

The main goal of the thesis was to give a comprehensive presentation of programming in LINQ by providing source code examples, so that the thesis can be used as study material for other programmers. The thesis is intended for programmers who already have experience with programming in .NET Framework and the C# programming language.

The thesis consists of an introduction to C# 3.0 new language features, query architecture, standard query operators forming a standardized interface for creating queries as well as a LINQ - to- SQL component which is used to perform object-relational mapping against a relational database and utilizing it as a data source for queries.

Keywords	LINQ, LINQ to SQL, C# 3.0, .NET Framework 3.5, object-relational mapping

Sisällys

Tiivistelmä
Abstract

1 Johdanto 6

2 LINQ-hakuarkkitehtuuri 7

2.1 Tietolähteet ohjelmoinnissa 7
2.2 Hakuarkkitehtuurin integroituminen ohjelmointikieliin 8
2.3 Hakujen toiminta 10
2.4 Komponentit 11

3 C# 3.0 -ohjelmointikielen uudet ominaisuudet 13

3.1 Implisiittisesti tyypitettävä paikallinen muuttuja ”var” 13
3.2 Olioiden ja kokoelmien alustajat 14
3.3 Anonyymit tyypit 15
3.4 Lambda-lausekkeet 16
3.5 Partial-metodit 17
3.6 Extension-metodit 19
3.7 Lausekepuut 20
3.8 Hakusyntaksi 21

4 Hakujen muodostaminen standardihakuoperaattoreilla 24

4.1 IEnumerable<T>-rajapinta 24
4.2 Viivästetyt ja ei-viivästetyt haut 24
4.3 Func-tyyppisen delegaatin käyttö standardihakuoperaattoreiden kanssa 27

5 Viivästetyt standardihakuoperaattorit 29

5.1 Rajoitusoperaatiot 29
5.2 Projektiot 30
5.3 Ositusoperaatiot 32
5.4 Järjestysoperaatiot 34
5.5 Joukko-operaatiot 36
5.6 Liitosoperaatiot 38
5.7 Ryhmittelyoperaatiot 42
5.8 Generointioperaatiot 44
5.9 Muunnosoperaatiot 46
5.10 Yhdistämisoperaatiot 48

6 Ei-viivästetyt standardihakuoperaattorit 49

6.1 Määritysoperaatiot 49
6.2 Yhdenvertaisuusoperaatiot 50
6.3 Elementtioperaatiot 51
6.4 Muunnosoperaatiot 53

6.5 Koosteoperaatiot 55

7 Linq To SQL 58

7.1 Esimerkkitietomalli ja vastaava esimerkkitietokanta 60
7.2 Olio-relaatiomallinnus 62
7.3 Tietomallin lähdekoodin generointi 64
7.4 DataContext-luokka 68
7.5 Tietokantaoperaatiot 72
7.6 Tietomallin toiminnallisuuden laajentaminen 79
7.7 Samanaikaisuuden hallinta ja konfliktien ratkaisu 82
7.8 Linq To SQL:n käyttö N-tasoarkkitehtuurissa 88

8 Yhteenveto 91

Lähteet 93

Liitteet

Liite 1: Standardihakuoperaattorien esimerkkien yhteinen lähdekoodi 95
Liite 2: Taulukko standardihakuoperaattoreista 97

1 Johdanto

Microsoft .NET Framework 3.5 -ohjelmistokehyksen ja C# 3.0 -ohjelmointikielen version julkaisun yhteydessä julkaistiin ”Language Intergrated Query” -niminen komponentti, lyhyemmin LINQ.

Komponentti tarjoaa ohjelmointikieleen täysin integroituvan hakuarkkitehtuurin, jolla voidaan hakea ohjelmallisesti tietoa yhtenäisen hakurajapinnan kautta mistä tahansa tietolähteestä, kuten relaatiotietokannasta, muistissa olevista oliokokoelmista tai XML tiedostoista.

Tässä insinöörityössä esitellään hakuarkkitehtuurin keskeinen toiminta ja standardihakuoperaattorit, jotka muodostavat yhtenäisen rajapinnan hakujen muodostamiseen eri tietolähteitä vasten.

Työn alussa esitellään C# 3.0 -ohjelmointikielen uudet ominaisuudet, joista suurin osa on kehitetty pelkästään mahdollistamaan hakuarkkitehtuurin toteuttaminen ja sen integroiminen ohjelmointikieleen ja joiden hallitseminen helpottaa huomattavasti sen toiminnan ymmärtämistä. Lopuksi esitellään konkreettisempi ja ehkä yleisimmin käytetty hakuarkkitehtuurin komponentti Linq To SQL, joka käyttää hakujen tietolähteenä relaatiotietokantaa.

Työssä esitellään lähdekoodiesimerkkejä, jotka kaikki ovat kirjoitettu C#-ohjelmointikielellä. Vaikka LINQ toimii myös Visual Basic -ohjelmointikielen kanssa, käytetään esimerkeissä pelkästään C#-kieltä. Lähdekoodiesimerkkien suorittamiseen tarvitaan Microsoft .NET 3.5 -ohjelmistokehys ja Microsoft Visual Studio 2008 -kehitysympäristö.

Työssä käsitellään C#-ohjelmointikieltä ja .NET-ohjelmistokehystä niiltä osin kuin se liittyy hakuarkkitehtuurin, joten työ vaatii lukijaltaan jo aikaisempaa tietämystä näiden käytöstä.

2 LINQ-hakuarkkitehtuuri

2.1 Tietolähteet ohjelmoinnissa

Ohjelmointi on kehittynyt viime vuosikymmenten aikana olio-ohjelmoinnin suuntaan, joten nykyään kaikille ohjelmoijille ja useimmille ohjelmointikielille on luonnollista tiedon käsittely olioina. Ongelmia olio-ohjelmoinnissa aiheuttaa tiedonhaku sellaisista tietolähteistä, jotka eivät käytä oliopohjaisia tekniikoita, kuten relaatiotietokannat ja XML.

Seuraavat seikat vaikeuttavat erilaisten tietolähteiden käyttöä ohjelmoinnissa:

- tietolähteiden käyttämät, toisistaan eroavat hakusyntaksit
- tietolähteiden toisistaan eroavat ohjelmointirajapinnat
- hakulausekkeiden käännöksen aikainen syntaksintarkistus.

Nykyään ohjelmoijan on osattava ulkoa lukuisten erilaisten tietolähteiden hakusyntaksi, tunnettava useita erilaisia ohjelmointirajapintoja ja vastaavien tietolähteiden sisäistä toimintaa, jotta ohjelmoija pystyy tehokkaasti käyttämään erilaisia tietolähteitä kehitettävissä sovelluksissa.

Hyvänä esimerkkinä tiedonhaun ongelmallisuudesta voidaan käyttää relaatiotietokantoja ja tiedonhakua niistä perinteisillä SQL-lauseilla. Ohjelmoijan on osattava muodostaa SQL-kielellä tietokantahakuja ja osattava käyttää tarvittavia rajapintoja tietokantahakujen suorittamiseksi. SQL-tietokantahaut joudutaan yleensä sijoittamaan C#-ohjelmakoodin sekaan erillisiin muuttujiin, jolloin ei voida saada virheilmoituksia mahdollisista lausekkeiden syntaksivirheistä ohjelmakoodia käännettäessä, vaan vasta ohjelmakoodin suorituksen aikana. Vastaavasti XML-muotoista tietolähdettä käyttäessä ohjelmoijan on tunnettava täysin relaatiotietokannan käytöstä eroava ohjelmointirajapinta ja hakusyntaksi.

Microsoft otti näiden ongelmien ratkaisuun yleisemmän lähestymistavan, kuin pelkästään .NET Framework -ohjelmistokehyksen parantamisen kehittämällä parempia eri tietolähteiden käsittelyyn tarkoitettuja luokkia kehittämällä kaikille tietolähteille sopivan yleispätevän hakuarkkitehtuurin.

LINQ toteuttaa ohjelmointikielitasolla olevan oliopohjaisen hakuarkkitehtuurin, jolla voidaan hakea tietoa melkein kaikista mahdollisista tietolähteistä käyttäen yhtenäistä hakurajapintaa. Koska se integroituu täysin .NET-ohjelmistokehyksen tukemiin ohjelmointikieliin, sen haut voivat hyödyntää suoraan metadatan käyttöä, käännöksen

aikaista syntaksitarkistusta, vahvaa tyypitystä ja Visual Studio 2008 IntelliSense -ominaisuutta.

2.2 Hakuarkkitehtuurin integroituminen ohjelmointikieliin

Hakuarkkitehtuuri on suunniteltu toimimaan oliokokoelmien kanssa, joista haetaan tietoa käyttämällä standardihakuoperaattoreita. Standardihakuoperaattorit ovat joukko extension-metodeita, jotka määrittelevät yleisen hakurajapinnan tietolähdettä vasten.

Standardihakuoperaattoreita voidaan käyttää haun muodostamisessa joko perinteisellä metodisyntaksilla tai hakusyntaksilla, joka vastaa syntaksiltaan perinteistä SQL-kieltä.

Koska haut suoritetaan aina oliokokoelmaan vasten, pitää tietolähteen data mallintaa ensiksi olioiksi. Tämä tapahtuu siten, että tietolähteelle toteutetaan erillinen komponentti, joka toteuttaa standardihakuoperaattorien määrittelemän rajapinnan, esimerkiksi Linq To XML -komponentti mallintaa XML:n muotoisen tiedon XElement-ja XDocument-tyyppisiksi olioiksi.

Haku koostuu kolmesta vaiheesta:

1. haun kohteena olevan tietolähteen muodostuksesta
2. suoritettavan haun muodostuksesta
3. haun suorituksesta.

Seuraavissa kolmessa esimerkissä on esitelty ”Hello World” -ohjelma, joka on toteutettu käyttäen seuraavia hakuarkkitehtuurin komponentteja: Linq To Objects, Linq To XML ja Linq To SQL. Esimerkeillä pyritään demonstroimaan hakuarkkitehtuurin eri käyttötarkoituksia ja antamaan yleiskuvaa hakujen toiminnasta eri tietolähteiden kanssa.

Ensimmäisessä esimerkissä käytetään Linq To Objects -komponenttia, jonka hakujen tietolähteenä toimivat muistissa olevat oliokokoelmat.

Esimerkki 1: Hello World -ohjelma käyttämällä Linq To Objects -komponenttia

```
// 1. Tietolähteen muodostus
string[] hello = { "Hello .NET World!",
                    "Hello C# 3.0",
                    "Hello LINQ World!" };

// 2. Haun muodostus
IEnumerable<string> result = from s in hello
                             where s.Contains("LINQ")
                             select s;

// 3. Haun suoritus
foreach (string s in result)
    Console.WriteLine(s);
```

Esimerkin 1 alussa muodostetaan tietolähde, jona toimii nyt hello-merkkijonotaulukko. Seuraavaksi muodostetaan haku käyttäen hakusyntaksia ja lopuksi suoritetaan haku. Esimerkissä haetaan hello-merkkijonotaulukosta ”LINQ”-merkkijonon sisältävä merkkijonon. Esimerkki tulostaa suoritettaessa seuraavan merkkijonon:

```
Hello LINQ World!
```

Esimerkissä 2 käytetään Linq To XML -komponenttia, jonka tietolähteenä toimii XML-muotoinen tieto.

Esimerkki 2: Hello World -ohjelma käyttäen Linq To XML -komponenttia

```
// 1. Tietolähteen muodostus
XElement xml = XElement.Parse(@"
                    <HelloWorlds>
                      <HelloWorld value=""Hello .NET World!"" />
                      <HelloWorld value=""Hello LINQ World!"" />
                      <HelloWorld value=""Hello C# 3.0"" />
                    </HelloWorlds>");

// 2. Haun muodostus
IEnumerable<XElement> result = xml.Elements()
    .Where(x => x.Attribute("value").Value.Contains("LINQ"));

// 3. Haun suoritus
foreach (XElement element in result)
    Console.WriteLine( element.Attribute("value").Value );
```

Esimerkki tulostaa suoritettaessa seuraavan merkkijonon:

```
Hello LINQ World!
```

Esimerkissä 3 esitellään Hello World -ohjelma käyttäen Linq To SQL -komponenttia, joka käyttää tietolähteenä relaatiotietokantaa.

Esimerkki 3: Hello World -ohjelma käyttäen Linq To SQL -komponenttia

```
// 1. Tietolähteen muodostus
using (DemoDataContext data = new DemoDataContext())
{
    // 2. Haun muodostus
    IQueryable<Product> result = data.Products
        .Where(p => p.Name == "Ferrari");

    // 3. Haun suoritus
    foreach (Product p in result)
        Console.WriteLine("{0} - {1}$", p.Name, p.Price);
}
```

Esimerkki tulostaa suoritettaessa seuraavan merkkijonon:

```
Ferrari - 200000.00$
```

Esimerkeissä esiteltiin hakuja käyttäen kolmea toisistaan täysin eroavaa tietolähdettä: muistissa olevia olioita, XML:ää ja relaatiotietokantaa. Esimerkeistä voidaan huomata, että vaikka tietolähteet eroavat tekniseltä rakenteeltaan toisistaan, voidaan niihin kaikkiin soveltaa samanlaisia hakuja. Samoin kaikkien esimerkkien tiedonhaun rakenne on yhtäläinen. Ensiksi muodostetaan tietolähde, sitten haku, joka lopuksi suoritetaan.

Esimerkeistä voidaan myös todeta hakujen integroituminen suoraan C#-kieleen. Missään esimerkissä ei tarvinnut kirjoittaa erillistä hakulausetta jollain muulla kielellä tai syntaksilla, ja kaikissa esimerkeissä käytetään vahvaa tyypitystä.

2.3 Hakujen toiminta

Haut suoritetaan aina oliota tai kokoelmaa vasten, joka toteuttaa IEnumerable<T>- tai IQueryable<T>-rajapinnan. Esimerkiksi kaikki .NET-ohjelmistokehyksen geneeriset kokoelmat toteuttavat IEnumerable<T>-rajapinnan, kuten List<T>-luokka ja abstrakti Array-luokka, joka toimii pohjana kaikille taulukoille.

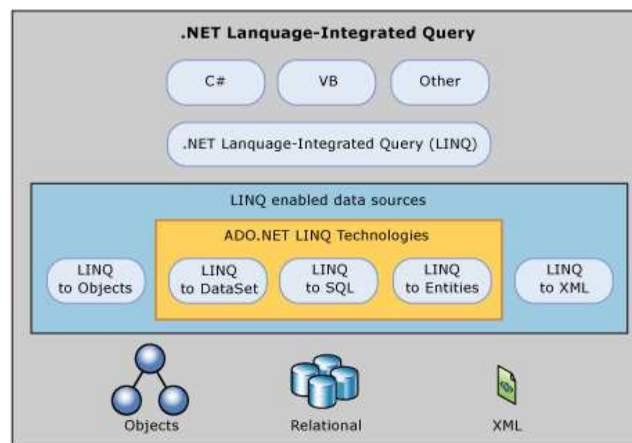
Riippuen haun kohteena olevan olion toteuttaman rajapinnan tyypistä haut joko suoritetaan hakuarkkitehtuurin sisäisellä hakumoottorilla tai hausta muodostetaan lausekepuu, joka annetaan haun kohteena olevalle tietolähteelle suoritettavaksi. Lausekepuut esitellään tarkemmin luvussa 3.7.

Haut, joiden kohteena olevat tietolähteet toteuttavat IEnumerable<T>-rajapinnan, suoritetaan käyttäen sisäistä hakumoottoria, ja IQueryable<T>-rajapinnan toteuttaviin tietolähteisiin kohdistuvat haut annetaan tietolähteelle suoritettaviksi.

Suoritettavien lausekkeiden muodostaminen ohjelmallisesti käsiteltäviksi tietorakenteiksi mahdollistaa yhdessä standardihakuoperaattorien määrittämän yhteisen rajapinnan kanssa hakuarkkitehtuurin laajentamisen mihin tahansa tietolähteeseen.

2.4 Komponentit

Kuvassa 1 on esitetty .NET-ohjelmistokehyksen mukana tulevat hakuarkkitehtuurin komponentit.



Kuva 1: .NET-ohjelmistokehykseen kuuluvat LINQ komponentit

Kuvassa keltaisella laatikolla ympäröidyt komponentit keskittyvät tiedonhakuun relaatiotietokannoista.

Linq To Objects -komponentti muodostaa hakuarkkitehtuurin ytimen. Linq To Objects -nimeä käytetään IEnumerable<T>-rajapinnasta, joka mahdollistaa hakujen suorittamisen muistissa olevia olioita vasten standardihakuoperaattorien avulla.

Linq To Objects kuuluu System.Core.dll kirjastoon, joten se on mukana automaattisesti kaikissa .NET 3.5 -ohjelmistokehystä käyttävissä projekteissa. Linq to Objects sijaitsee System.Linq nimiavaruudessa.

Linq To XML -komponentti on oliopohjainen toteutus XML-muotoisen tiedon käsittelyyn, joka mahdollistaa muun muassa dokumenttien luomisen, muokkaamisen, tallentamisen ja tiedon hakemisen dokumenteista käyttäen standardihakuoperaattoreita.

Linq To XML eroaa DOM (Document Object Model) -mallisesta XML-tiedon käsittelystä sillä, että se keskittyy enemmänkin dokumenttien osien käsittelyyn, siinä missä DOM-mallissa käsitellään aina kokonaista muistiin ladattua dokumenttia. Linq to XML myös mahdollistaa XSL-muunnosta vastaavan dokumenttien funktionaalisen muunnoksen käyttämällä standardihakuoperaattoreita, ilman että muunnettava dokumentti ladataan

kokonaisuudessaan muistiin. Näin ollen sen suorituskyky XSL-muunnoksissa on huomattavasti parempi kuin DOM-mallia käytettäessä.

Linq To SQL -komponentti käyttää hakujen tietolähteen Microsoft SQL Server -tietokantaa, ja se myös toteuttaa tietokannalle olio-relaatiomallinnuksen.

Linq To SQL on toteutettu IQueryable<T>-rajapinnan kautta, jossa hausta muodostetaan SQL-lause, joka annetaan tietokannan suoritettavaksi. Tietokantahausta saadut tulokset muunnetaan takaisin oliomuotoon jatkokäsittelyä varten.

Linq To DataSet -komponentti mahdollistaa hakujen suorittamisen perinteisiä ADO.NET DataSet- ja DataTable-luokkia vasten. Se laajentaa luokkien toimintaa siten, että niitä vasten voidaan suorittaa standardihakuoperaatioita. Tämä mahdollistaa hakujen hyödyntämisen projektissa, joka käyttää perinteistä ADO.NET-tekniikkaa relaatiotietokannan tiedonhakuun.

Entity Framework on olio-relaatiomallinnus ohjelmointikehys, joka vastaa toiminnaltaan Linq To SQL komponenttia, mutta on osaltaan monimutkaisempi ja mahdollistaa tarkemman olio-relaatiomallinnuksen relaatiotietokantaa vasten.

Entity Framework sisältää **Linq To Entities** -komponentin, joka mahdollistaa olioiden hakemisen tietokannasta mallinnetusta tietomallista. Entity Framework muuntaa haut tietokannasta riippumattomiksi Entity SQL -lausekkeiksi, jotka muunnetaan tietokantatarjoajan tietokannan toteutuksesta riippuviksi hakulauseiksi hakujen suorituksen yhteydessä. Näin ollen se tukee useita eri valmistajien tietokantoja, jotka ovat toteuttaneet Entity Frameworkille muunnoskomponentin.

Kolmansien osapuolien komponentit

Toteuttamalla mille tahansa tietolähteelle IEnumerable<T>- tai IQueryable<T>-rajapinnat, on mahdollista kehittää uusia tietolähteitä, jotka hyödyntävät hakuarkkitehtuuria tiedonhaussa. Useat kolmannet osapuolet ovat jo toteuttaneet tietolähteilleen hakuarkkitehtuuria hyödyntäviä komponentteja, kuten esimerkiksi Linq To Active Directory -projekti, joka käyttää hakujen tietolähteen Active Directory -käyttäjätietokantaa. [14.]

3 C# 3.0 -ohjelmointikielen uudet ominaisuudet

C#-ohjelmointikielen version 3.0 uusien ominaisuuksien kehittämisen pohjana on toiminut hakuarkkitehtuurin toteuttamisen mahdollistaminen ja sen integroiminen mahdollisimman hyvin ohjelmointikieleen, siten että hakujen muodostaminen olisi yksikertaista ja helposti omaksuttavaa. Osa uusista ominaisuuksista, kuten lambda-lausekkeet ja extension-metodit, ovat myös hyödyllisiä muussakin ohjelmointikäytössä kuin pelkästään hakujen muodostamisessa. [17.]

```
var result1 = from p in db.Products
              where p.Price > 10000
              select new { Name = p.Name, p.Price };

var result2 = db.Products
              .Where(p => p.Price > 10000)
              .Select(p => new { Name = p.Name, p.Price });
```

Kuva 2: C# ohjelmointikielen uusien ominaisuuksien liittyminen hakujen muodostukseen.

Kuvassa 2 havainnollistetaan, miten C#-ohjelmointikielen uudet ominaisuudet liittyvät hakujen muodostamiseen. Kuvaan on merkitty numeroilla seuraavat ohjelmointikielen uudet ominaisuudet, jotka esitellään myöhemmin:

1. hakusyntaksi
2. Implisiittisesti tyypitettävä paikallinen var muuttuja
3. lambda-lauseke
4. extension-metodi
5. anonyymin tyyppi
6. olion alustaja.

3.1 Implisiittisesti tyypitettävä paikallinen muuttuja ”var”

Implisiittisesti tyypitettävällä muuttujalla tarkoitetaan muuttujaa, jolle ei erikseen määritellä tietotyyppiä, vaan kääntäjä päättelee muuttujan tyypin siihen sijoitettavan arvon tyypin perusteella. Tällainen muuttuja määritellään var-avainsanalla esimerkin 4 tavoin.

Esimerkki 4: Var-muuttujan esittely ja initialisointi.

```
var name = ”Test”;
```

Var-muuttujaa ei voida esitellä ilman initialisointia, koska kääntäjän pitää pystyä selvittämään muuttujan tyyppi siihen sijoitettavasta arvosta. Muuttuja on myös vahvasti tyypitetty, eli kääntäjä tarkistaa muuttujaan sijoitettavien arvojen oikean tyypin ja antaa virheilmoituksen virheellisistä sijoituksista.

Vaikka var-muuttujaan on mahdollista sijoittaa mitä tahansa arvoja, ja rajoitteena on vain sijoitettavan arvon kunnollinen initialisointi, ei sitä kannata ohjelmakoodin luettavuuden takia käyttää kuin sitä vaativissa tilanteissa. Näitä tilanteita ovat esimerkiksi anonyymien tyyppien käyttö ja standardihakuoperaattorien projektiot.

3.2 Olioiden ja kokoelmien alustajat

Olion alustajalla tarkoitetaan olion luomista ja sen ominaisuuksien asettamista samalla ohjelmakoodilausekkeella. Olioiden ja kokoelmien alustajat luotiin C#-kieleen ratkaisemaan standardihakuoperaattoreiden projektioiden tarve luoda olio ja asettaa halutut arvot sen ominaisuuksiin. Ilman alustajia projektiossa luotavalla oliolle pitäisi tehdä ylikirjoitettu rakennin jokaista erilaista asetettavien ominaisuuksien variaatiota varten, samoin anonyymien tyyppien luominen olisi mahdotonta ilman olioiden alustajia. [17.]

Esimerkki 5: User-olion luominen käyttämällä olion alustajaa.

```
User user = new User { Name = "Mikko", Age = 23 };

Console.Write("ID: {0}\nName: {1}\nAge: {2}",
    user.ID, user.Name, user.Age);
```

Esimerkissä 5 esitellään User-olion luominen ja sen Name- ja Age-ominaisuuksien asettaminen käyttämällä olion alustajaa.

Olion alustuksen syntaksissa jätetään pois sulkumerkit ( ), jotka normaalisti tarkoittaisivat olion rakentimen kutsua, ja tehdään haluttujen ominaisuuksien asettaminen kaarisulkujen sisällä.

Olion alustaja luo luokan instanssin käyttämällä oletusrakenninta, jonka jälkeen asetetaan olion julkiset ominaisuudet. Ne ominaisuudet, joita ei erikseen aseteta, saavat oletusarvon.

Kokoelmien alustajat

Kokoelmien alustajat toimivat samalla tavalla kuin olioiden alustajat, eli ne mahdollistavat kokoelman arvojen asettamisen kokoelman luomisen yhteydessä. Kokoelmien alustajat

toimivat kaikilla geneerisillä kokoelmilla, jotka toteuttavat ICollection<T>-rajapinnan. Esimerkissä 6 on esitelty kokoelman alustajan käyttö listan luonnissa.

Esimerkki 6: List<User>-tyyppisen kokoelma luonti käyttämällä kokoelman alustajaa.

```
List<User> userList = new List<User>
    {
        new User{ Name="Mikko", Age=23 },
        new User{ Name="Sami" }
    };
```

3.3 Anonyymit tyypit

Anonyymit tyypit mahdollistavat vahvasti tyypitettyjen olioiden määrittämisen dynaamisesti.

Jotta hakutuloksista olisi mahdollista projisoida vain halutut tiedot vahvasti tyypitettyinä olioina, pitää palautettavan olion tyyppi olla määritelty. Ilman mahdollisuutta määritellä näitä tyyppejä dynaamisesti pitäisi jokaista erilaista projektio-operaatiota varten määritellä erikseen operaation palauttamaa tietoa vastaava luokka, mikä olisi huomattavan työlästä. Anonyymit tyypit kehitettiin C#-kieleen juuri ratkaisemaan tämän ongelman. [17.]

Anonyymeillä tyypeillä ei ole nimeä, vaan kääntäjä generoi tyypin perustuen anonyymin olion initialisointiin, minkä johdosta anonyymiä tyyppiä ei voida sijoittaa tyypitettyyn muuttujaan, vaan sijoitus onnistuu vain var-muuttujaan.

Anonyymi tyyppi luodaan samoin kuin mikä tahansa muukin tyyppi, mutta tyypin nimi jätetään luonnissa pois ja sen ominaisuudet asetetaan käyttäen olion alustajaa. Anonyymin tyypin asetettavat ominaisuudet ovat vain luettavissa olion luonnin jälkeen.

Esimerkki 7: Anonyymin tyypin luominen.

```
// Anonyymin tyypin luonti
var person = new
  {
    Name = "Mikko",
    Age = 23
  };

Console.WriteLine(string.Format("Name: {0}, Age: {1}",
        person.Name, person.Age));

// Tulostetaan kääntäjän generoiman tyypin nimi:
Console.WriteLine(person.GetType());
```

Esimerkissä 7 on esitelty anonyymin tyypin luonti, jossa asetetaan tyypille kaksi ominaisuutta, Name ja Age, ja tulostetaan kääntäjän generoiman tyypin nimi.

Jos samassa muuttujien näkyvyysalueessa määritellään kaksi täysin samat ominaisuudet omaavaa anonyymiä tyyppiä, tulkitsee kääntäjä ne samaksi tyypiksi ja generoi vain yhden vastaavan tyyppimäärityksen.

Anonyymit tyypit periytyvät aina Object-luokasta, ja sen Equals- ja GetHashCode-metodien toteutus perustuu tyypin ominaisuuksien vastaavien metodien toteutukseen. Näin ollen anonyymien tyyppien instanssit ovat samat vain, jos niiden kaikki ominaisuudet ovat samat.

3.4 Lambda-lausekkeet

Lambda-lausekkeet vastaavat monin tavoin C#-kielen anonyymejä metodeja ja mahdollistavat algoritmien kirjoittamisen tiivistetysti omalla syntaksilla. Ne kehitettiin korvaamaan anonyymien metodien kömpelö syntaksi delegaattien määrittelyssä ja helpottamaan näin standardihakuoperaattoreiden käyttöä.

Lambda-lausekkeet määritellään alkaen pilkulla erotellulla parametrilistalla, jota seuraa lambda-operaattori ja lausekeblokki. Esimerkissä 8 on kirjoitettu lambda-lauseke yhdellä rivillä, joka voidaan sanallisesti tulkita seuraavasti: ”syötteet x ja y palauttavat tuloksen x + y”.

Esimerkki 8: Lambda lauseke yhdellä rivillä.

```
(x, y) => x + y
```

Lambda-lausekkeet voidaan kirjoittaa myös käyttäen normaaleja C#-lausekkeita ja käyttäen pidempää syntaksia, mikä helpottaa monimutkaisten lausekkeiden luettavuutta. Tällainen isompi lausekelohko määritellään kaarisulkujen sisään, kuten normaalit ohjelmakoodilohkotkin. Esimerkissä 9 on esitelty pidempi lambda-lauseke. Kyseinen lauseke ottaa parametrikseen kaksi arvoa ja palauttaa niistä suuremman tai nollan, jos arvot ovat yhtä suuret.

Esimerkki 9: Pidempi lambda-lauseke jossa useita lausekkeita.

```
(x, y) =>
{
    if (x > y)
    {
        return x;
    }
    else if (x == y)
    {
        return 0;
    }
    else
    {
        return y;
    }
}
```

Jos lambda-lauseke saa syötteenä vain yhden parametrin, ei parametrin ympärillä tarvita sulkuja.

Aikaisemmista esimerkeistä ei vielä selviä, minkätyyppisiä parametreja lambda-lausekkeille pitää antaa tai mikä on niiden palautusarvon tyyppi. Tämän määrittää delegaatin tyyppi, jossa lambda-lauseketta käytetään. Määritellään esimerkiksi seuraavanlainen delegaatti.

```
delegate int Calculate(int i, int k);
```

Edellä oleva Calculate-delegaatti ottaa parametrikseen kaksi kokonaislukua ja palauttaa kokonaisluvun, joten vastaavalle delegaatille määriteltävän lambda-lausekkeen pitää ottaa kaksi kokonaislukuparametria ja palauttaa kokonaisluku.

Kääntäjä tarkistaa lambda-lausekkeiden parametrien ja palautusarvon tyyppien oikeellisuuden, eikä salli virheellisesti tyypitettyjen lausekkeiden sijoittamista delegaatteihin.

Silloin kun lambda-lausekkeella toteutettava algoritmi on monimutkainen tai samaa algoritmia käytetään toistuvasti, kannattaa se toteuttaa nimettynä metodina.

3.5 Partial-metodit

C# 2.0 -ohjelmointikielessä esiteltiin partial-luokat eli luokat, jotka on mahdollista määritellä useassa eri lähdekooditiedostossa ja jotka kääntämisen yhteydessä yhdistetään yhdeksi luokkamäärittelyksi. C# 3.0 -ohjelmointikielessä tätä toiminnallisuutta laajennettiin koskemaan metodien määrittelyä. Partial-metodit ovat käytännössä kevyitä luokan sisäisiä tapahtumankäsittelijöitä, joihin ei erikseen tarvitse liittää delegaatteja.

Partial-metodit voidaan määritellä vain partial-luokalle tai tietueelle. Ne koostuvat kahdesta osasta, metodin esittelystä ja metodin toteutuksesta, ja mahdollistavat metodin esittelyn ja kutsumisen yhdessä osassa luokan määritystä ja toteutuksen toisessa osassa. Tällöin sitä voidaan käyttää tapahtuman tavoin.

Partial-metodille ei tarvitse välttämättä tehdä toteutusta, jolloin kääntäjä poistaa kääntämisen yhteydessä ne metodien kutsut ja esittelyt, joille ei ole vastaavaa toteutusta. Tämän toiminnallisuuden johdosta voidaan määritellä luokalle useita laajennettavia osia, ilman että ohjelmakoodin suorituskyky hidastuu.

Partial-metodeita koskevat seuraavat rajoitteet:

- Metodi voi esiintyä vain partial luokassa.
- Metodi pitää määritellä ”partial” määreellä ja metodilla ei voi olla palautusarvoa.
- Metodit ovat aina näkyvyydeltään yksityisiä metodeja.

- Metodeilla ei voi olla mitään näkyvyysmääreitä, kuten private, protected, ja public tai periytymismääreitä kuten esim. virtual tai abstact.
- Metodi voi olla staattinen metodi.

Esimerkki 10: Esimerkki partial-metodin käytöstä.

```
// Ensimmäinen osa luokkamääritystä
public partial class Person
{
    private string phoneNumber;
    public string PhoneNumber
    {
        get { return phoneNumber; }
        set
        {
            // 2. Partial metodin kutsu
            OnPhoneNumberChange(value);
            phoneNumber = value;
        }
    }

    // 1. Partial metodin esittely
    partial void OnPhoneNumberChange(string s);
}

// Toinen osa luokkamääritystä
public partial class Person
{
    // 3. Partial metodin toteutus
    partial void OnPhoneNumberChange(string newValue)
    {
        if (!Regex.IsMatch(newValue, "^\\d{3}-\\d{7}"))
            throw new ArgumentException("Not a phone number.");
    }
}
```

Esimerkissä 10 määritellään Person-luokka kahdessa eri osassa, jotka esimerkin yksinkertaistamiseksi ovat nyt allekkain, mutta luokkien määrittely voisi myös sijaita eri tiedostoissa. Person-luokalla on PhoneNumber-ominaisuus ja OnPhoneNumberChange-niminen partial-metodi, jota kutsutaan ominaisuuden asetuslohkosta. Metodille annetaan parametriksi ominaisuuteen sijoitettava arvo.

Luokan toisessa määrittelyosassa on tehty OnPhoneNumberChange-metodille toteutus, joka suorittaa sijoitettavalle arvolle arvon oikeellisuuden tarkistuksen.

Esimerkin havainnollistama partial-metodin käyttäminen luokkien ominaisuuksiin sijoitettavien arvojen oikeellisuuden tarkistamisessa on hyvin yleinen käyttötapaus partial-metodeille. Metodeja käytetään usein generoidun ohjelmakoodin yhteydessä, mahdollistamaan generoidun luokan toimintalogiikan laajentamisen, kuten Linq To SQL -komponentin käytössä.

3.6 Extension-metodit

Extension-metodit mahdollistavat uusien metodien lisäämisen jo olemassa oleviin luokkiin, ilman että olemassa olevaa luokkaa tarvitsee periyttää tai muuten muokata. Extension-metodit ovat staattisen luokan staattisia metodeja, joita voidaan kutsua toisen luokan instanssin metodeina.

Hakuarkkitehtuurin toiminta ja laajennettavuus perustuvat juuri extension-metodeihin. Yleisimmät extension-metodit ovat standardihakuoperaattorit, jotka laajentavat IEnumerable- ja IEnumerable<T>-luokkien toimintaa.

Extension-metodi voidaan määritellä vain staattisessa luokassa. Se määritellään samalla tavalla kuin normaali staattinen metodi, poikkeuksena se että metodin ensimmäiselle parametrille annetaan ”this”-määre ja ensimmäisen parametrin tyyppi määrää, mille luokalle metodi tulee käytettäväksi eli minkä luokan toiminnallisuutta metodi laajentaa.

Extension-metodin kutsu vastaa staattisen metodin kutsua, jonka ensimmäiseksi parametriksi välitetään referenssi kyseisen luokan instanssiin, jolla metodia kutsutaan.

Esimerkki 11: Extension-metodin toteutus.

```
namespace SampleExtensions
{
    public static class IntExtensions
    {
        public static bool IsOdd(this int i)
        {
            return i % 2 == 1;
        }
    }
}
```

Esimerkissä 12 on määritelty IsOdd-niminen extension-metodi, joka laajentaa int-tyypin toiminnallisuutta. Metodi palauttaa totuusarvon siitä, onko metodin saama int-tyypin instanssi arvoltaan pariton.

Jotta extension-metodeita saadaan käyttöön, täytyy ohjelmakoodiin liittää mukaan sama nimiavaruus, jossa sijaitsee käytettävien metodien luokkamääritykset.

Esimerkki 12: Extension-metodin kutsuminen.

```
using SampleExtensions;
...

int[] numbers = { 1, 2, 3, 4, 7, 48, 75 };

IEnumerable<int> odds = numbers.Where( number => number.IsOdd() );

foreach (int i in odds)
    Console.WriteLine(i);
```

Esimerkissä 12 on esitelty esimerkissä 11 määritellyn IsOdd extension-metodin käyttäminen. Esimerkissä haetaan kokonaislukutaulukosta vain parittomat luvut käyttämällä Where-standardihakuoperaattoria ja aikaisemmin määriteltyä IsOdd extension-metodia ehtolausekkeessa.

Kutsuttavan metodin selvityksessä käännöksen aikana ovat extension-metodit aina takasijalla verrattuna luokan normaaleihin metodeihin. Jos laajennettavalla luokalla on jo olemassa vastaavalla signatuurilla oleva metodi, eli metodi on samanniminen ja metodilla on samantyyppiset parametrit, ei kyseistä extension-metodia tulla koskaan kutsumaan.

Extension-metodit ovat erittäin käytännöllisiä silloin kun on tarve laajentaa jonkin luokan toimintaa, mutta kyseistä luokkaa ei ole mahdollista periä.

3.7 Lausekepuut

Lausekepuu on tehokas datan esitysmuoto, jossa data esitetään hierarkkisessa puumaisessa muodossa. Lausekepuun sisältämä data on nopea järjestää ja käydä läpi. Lausekepuuta kuvaa Expression<TDelegate>-luokka.

Expression<TDelegate>-luokalla voidaan esittää mikä tahansa delegaatti ohjelmallisesti käsiteltävänä tietorakenteena. Jos ohjelmakoodissa on sijoitusoperaatio Expression<TDelegate>-tyyppisen muuttujaan, muodostaa kääntäjä sijoitettavasta delegaatista automaattisesti lausekepuun, joka sisältää delegaatin suoritettavat lausekkeet. Tällöin lausekepuun vastaan ottava standardihakuoperaattori voi muodostaa ohjelmallisesti lausekepuun sisältämistä lausekkeista esimerkiksi vastaavan SQL-lauseen tietokantahakua varten.

Standardihakuoperaattoreissa suoritettavan metodin prototyyppi määrää, muodostetaanko annetusta delegaatista lausekepuu vai suoritetaanko se sellaisenaan. Jos standardihakuoperaattori ottaa parametrikseen delegaatin, muodostetaan normaali delegaatin kutsu ja jos operaattori ottaa vastaa Expression<TDelegate>-tyypin, niin muodostetaan lausekepuu. Esimerkiksi IQueryable<T>-rajapinta sisältää ylikuormitetun version IEnumerable<T>-luokan standardihakuoperaattoreista, jotka ottavat parametrikseen Expression<TDelegate>-tyypin.

Esimerkki 13: Lausekepuun muodostaminen ja sen käsittely ohjelmallisesti.

```
// using System.Linq.Expressions;
...
// Luodaan lausekepuu
Expression<Func<int, int>> expr = number => number * 2;

// Puretaan lausekepuu osiksi:
ParameterExpression inputParam = expr.Parameters[0];
BinaryExpression operation = (BinaryExpression) expr.Body;
ParameterExpression leftValue = (ParameterExpression) operation.Left;
ConstantExpression rightValue = (ConstantExpression) operation.Right;

// Tulostetaan
Console.WriteLine("Expression: {0} => {1} {2} {3}",
    inputParam.Name,
    leftValue.Name,
    operation.NodeType,
    rightValue.Value);

// Käännetään expression puu suoritettavaksi delegaatiksi
Func<int, int> deleg = expr.Compile();

// Suoritetaan käännetty expressio arvolla 2
Console.WriteLine("Result: {0}", deleg(2));
```

Esimerkissä 13 esitellään lausekepuun muodostaminen ja käsittely. Esimerkissä muodostetaan lambda-lausekkeesta, joka ottaa parametrikseen kokonaisluvun ja kertoo sen kahdella, lausekepuu. Ensiksi luodaan Expression<Func<int, int>>-tyyppinen luokka, joka kapseloi lausekepuun. Luotu lausekepuu paloitellaan osiin ja osien nimet tulostetaan luettavaan muotoon. Lopuksi käännetään lausekepuu suoritettavaksi delegaatiksi ja suoritetaan se.

Esimerkki tulostaa suoritettaessa seuraavan tulosteen:

```
Expression: number => number Multiply 2
Result: 4
```

Ensimmäisellä rivillä on lausekepuun osat tulostettuna luettavaan muotoon, josta näkee, että lausekepuu vastaa annettua lambda-lauseketta. Toisella rivillä nähdään lausekepuusta käännetyn delegaatin suorituksen lopputulos.

3.8 Hakusyntaksi

C# 3.0 -ohjelmointikielen yhtenä uutena ominaisuutena on hakusyntaksi, jota voidaan käyttää pelkästään sellaisenaan hakujen muodostamiseen, tai metodisyntaksin lisänä. Hakusyntaksi muistuttaa paljolti SQL-kieltä, ja se parantaa monien hakujen luettavuutta varsinkin silloin kun ohjelmoijalla on taustaa SQL osaamisesta. [17.]

Hakusyntaksi koostuu joukosta lausekkeita, jotka kääntäjä kääntämisen yhteydessä muuntaa standardihakuoperaattoreiden metodikutsuiksi.

Esimerkissä 14 on esitelty hakusyntaksin käyttäminen.

Esimerkki 14: Hakusyntaksin käyttö

```
int[] numbers = { 0, 1, 2, 3, 4, 5 };

IEnumerable<int> result = from n in numbers
                          where n >= 3
                          orderby n descending
                          select n;

foreach (int i in result)
    Console.Write("{0} ", i);
```

Esimerkin haku hakee numbers-taulukosta ne numerot, jotka ovat yhtä suuria tai suurempia kuin kolme, ja järjestää hakutuloksen numerot laskevaan järjestykseen.

Esimerkissä 15 on esitelty edellisen esimerkin hausta kääntäjän muodostama ohjelmakoodi.

Esimerkki 15: Kääntäjän muodostama ohjelmakoodi hakusyntaksista.

```
IEnumerable<int> result = numbers.Where<int>(delegate (int n) {
    return (n >= 3);
}).OrderByDescending<int, int>(delegate (int n) {
    return n;
});
```

Esimerkistä 15 nähdään, että kääntäjä muuntaa hakusyntaksilla tehdyn haun standardihakuoperaattoreiden metodikutsuiksi. Esimerkistä voidaan nähdä, että ”where”-lauseke on muuttunut Where-standardihakuoperaattorin kutsuksi ja ”orberby n descending”-lauseke on muuttunut OrderByDescending-standardihakuoperaattorin kutsuksi.

Hakusyntaksin säännöt

Hakusyntaksilla tehtyjen hakulausekkeiden pitää noudattaa seuraavia sääntöjä: [1, s. 37–40; 8.]

1. Hakulausekkeen pitää alkaa from-lausekkeella.
2. Ensimmäisen from-lauseen ja viimeisen select- tai group-lauseen välissä voi olla yksi tai useampi seuraavia lausekkeita: where, orderby, join, let ja from.
3. Haun pitää päättyä joko select- tai group-lausekkeeseen.
4. Hakua voidaan jatkaa into-lausekkeella, jolloin edellisen haun tulokset toimivat seuraavan haun lähteenä.

Kääntäjä tarkistaa hakusyntaksin syntaksivirheet kääntämisen yhteydessä.

Kaikille standardihakuoperaattoreille ja operaattorien kombinaatioille ei löydy vastaavaa hakusyntaksia, minkä vuoksi joudutaan käyttämään metodisyntaksia. Hakusyntaksia ja metodisyntaksia voidaan myös käyttää ristiin, mutta tämä usein tekee ohjelmakoodista vaikeaselkoista. Ohjelmakoodin toimivuuden kannalta ei ole suurta eroa siinä, kumpaa

syntaksia käyttää. Enemmän se vaikuttaa ohjelmakoodin luettavuuteen, joten on suositeltavaa käyttää sitä syntaksia, joka tuottaa helpoiten ymmärrettävän ohjelmakoodin. [2, s. 44.]

4 Hakujen muodostaminen standardihakuoperaattoreilla

Tässä luvussa esitellään hakujen muodostaminen standardihakuoperaattoreilla. Luvun keskeisiä asioita ovat IEnumerable<T>-rajapinta, viivästetyt haut ja Func-delegaatin käyttäminen standardihakuoperaattoreiden kanssa.

4.1 IEnumerable<T>-rajapinta

Linq To Objects mahdollistaa hakujen suorittamisen muistissa olevia oliokokoelmia vasten, jotka toteuttavat IEnumerable<T>-rajapinnan. Näihin kokoelmiin kuuluvat kaikki geneeriset kokoelmat, kuten esimerkiksi taulukot.

Rajapinta mahdollistaa kokoelman elementtien listauksen, eli toisin sanoen IEnumerable<T>-rajapinnan toteuttava kokoelma on tyyppiä T olevien elementtien sekvenssi. Esimerkiksi IEnumerable<int>-tyyppisestä kokoelmasta voidaan sanoa, että se on sekvenssi kokonaislukuja.

Suurin osa standardihakuoperaattoreista on toteutettu IEnumerable<T>-tyypin extension-metodeina staattisessa Enumerable-luokassa, ja ne ottavat ensimmäiseksi parametrikseen IEnumerable<T>-tyyppisen olion ja yleensä palauttavat IEnumerable<T>- tai IOrderedEnumerable<T>-tyyppisen sekvenssin, poikkeuksena ovat ei-viivästetyt haut.

Koska standardihakuoperaattorit palauttavat sekvenssejä, on mahdollista muodostaa kompleksisia hakuja kutsumalla uutta standardihakuoperaattoria edellisen palautusarvolle.

4.2 Viivästetyt ja ei-viivästetyt haut

IEnumerable<T>-tyyppisen olion palauttavat standardihakuoperaattorien muodostamat haut suoritetaan tietolähdettä vasten vasta siinä vaiheessa kun haun palauttama sekvenssi listataan läpi. Standardihakuoperaattorien palauttama sekvenssi ei siis suoraan sisällä haettuja oliota, vaan se tuottaa T-tyyppisen olion sekvenssin läpikäynnin aikana. Tätä kutsutaan ns. viivästyneeksi hauksi. Viivästetyt haut mahdollistavat kompleksisten hakujen muodostamisen ja tukevat funktionaalista ohjelmointimallia.

Koska haut suoritetaan viivästyneesti, haun mahdollisesti aiheuttama poikkeus tapahtuu vasta haun palauttaman sekvenssin listauksen yhteydessä. Esimerkissä 16 esitellään viivästyneen haun suoritus ja poikkeus haussa.

Esimerkki 16: Poikkeus viivästyneen haun suorituksessa.

```
int[] numbers = { 1, 2, 3, 4, 5 };
string[] strings = { "if", "then", null, "while", "foreach" };

IEnumerable<int> result1 = numbers.Where(i => i > 2);

// Ei poikkeusta vielä tässä
IEnumerable<string> result2 = strings.Where(s => s.Length > 3);

try
{
    // Listataan result1
    foreach (int i in result1)
        Console.WriteLine(i);

    // Listataan result2
    foreach (string s in result2)
        Console.WriteLine(s);
}
catch (Exception e)
{
    Console.WriteLine("{0} - {1}", e.GetType(), e.Message );
}
```

Esimerkissä 16 määritellään kaksi taulukkoa, numbers ja strings, joista strings-taulukkoon on sijoitettu null-arvo, joka aiheuttaa poikkeuksen haun suorittamisessa. Esimerkissä itse hakujen suoritus on sijoitettu try-catch-lohkoon, jolla havainnollistetaan viivästetyn haun suoritusajankohtaa ja saadaan aiheutettu poikkeus kiinni. Esimerkki tulostaa suoritettaessa seuraavan tulostuksen:

```
3
4
5
then
System.NullReferenceException - Object reference not set to an instance of
an object.
```

Esimerkin tulosteesta voidaan huomata, että result1-sekvenssin läpikäynti onnistui odotetusti ja result2-sekvenssin läpikäynnissä ”then”-arvon jälkeinen null-arvo aiheutti poikkeuksen. Huomioitavaa on myös se, että poikkeus tosiaan aiheutuu vasta standardihakuoperaattorin palauttaman sekvenssin läpikäynnin aikana, eikä silloin kun operaattoria kutsutaan.

Hakujen suorittaminen vasta haun palauttaman sekvenssin läpikäynnin yhteydessä mahdollistaa yhden haun suorittamisen useampaan kertaan läpikäymällä haun palauttama sekvenssi useasti. Jos haun kohteena olevan tiedon sisältö muuttuu sekvenssin läpikäyntien välissä, saadaan myös eri hakutulokset. Esimerkissä 17 esitellään saman haun suorittaminen kahteen kertaan ja lähdetiedon muuttuminen hakujen välissä.

Esimerkki 17: Hakutulosten muuttuminen viivästettyjen hakujen välissä.

```
int[] numbers = { 0, 1, 2, 3 };

IEnumerable<int> result = numbers.Select(i => i);

// Suoritetaan haku ensimmäisen kerran
foreach (int i in result)
    Console.WriteLine(i);

Console.WriteLine("---");

// Muutetaan lähdedataa
numbers[0] = 5;
numbers[2] = 6;

// Suoritetaan haku toisen kerran
foreach (int i in result)
    Console.WriteLine(i);
```

Esimerkissä 17 suoritetaan sama haku kahteen kertaan ja muutetaan lähdetietoa hakujen välillä. Esimerkin tulostaa suoritettaessa seuraavan tulostuksen:

```
0
1
2
3
---
5
1
6
3
```

Jos haku halutaan suorittaa heti esimerkiksi silloin kun hakutulos halutaan tallentaa välimuistiin jälkikäsittelyä varten, voidaan siihen käyttää ei-viivästettyjä standardihakuoperaattoreita.

Ei-viivästetyn standardihakuoperaattorin tunnistaa siitä, että se ei palauta IEnumerable<T>- tai IOrderedEnumerable<T>-tyyppistä oliota.

Esimerkki 18: Listan palautus ToList-operaattorilla, jolloin haku suoritetaan välittömästi.

```
int[] numbers = { 0, 1, 2, 3 };

// Haku suoriteaan välittömästi
List<int> result = numbers.Select(i => i).ToList();

// Listataan hakutulos ensimmäisen kerran
foreach (int i in result)
    Console.WriteLine(i);

Console.WriteLine("---");

// Muutetaan lähdedataa
numbers[0] = 5;
numbers[2] = 6;

// Listataan hakutulos toisen kerran
foreach (int i in result)
    Console.WriteLine(i);
```

Esimerkki 18 vastaa aiemmin esiteltyä esimerkkiä 17, jossa suoritettiin sama haku viivästyneesti kahteen kertaan ja niiden välissä muutettiin lähdedataa. Esimerkki eroaa siten, että nyt haku suoritetaan välittömästi käyttäen ToList-standardihakuoperaattoria ja haun tulokset tallennetaan List<int>-tyyppiseen kokoelmaan. Tällöin hakutulosten läpikäynti tuottaa täsmälleen saman tuloksen, vaikka lähdetieto on muuttunut välissä. Esimerkki tulostaa suoritettaessa seuraavan tulostuksen:

```
0
1
2
3
---
0
1
2
3
```

4.3 Func-tyyppisen delegaatin käyttö standardihakuoperaattoreiden kanssa

Monet standardihakuoperaattorit saavat parametrikseen yhden tai useamman geneerisen Func-tyyppiä olevan delegaatin, joka on määritelty yhtenäistämään standardihaku-operaattoreiden käyttöä ja helpottamaan delegaattien kirjoittamista lambda-lausekkeilla.

.NET 3.5 -ohjelmistokehys sisältää Func-delegaatista viisi eri määrittelyä, jotka eroavat toisistaan vain parametrien määrällä. Delegaatin eri määrittelyt on listattu esimerkissä 19.

Esimerkki 19: .NET 3.5 ohjelmistokehyksen sisältämät Func-delegaatin määrittelyt.

```
public delegate TResult Func<TResult>()

public delegate TResult Func<T, TResult>(
    T arg
)

public delegate TResult Func<T1, T2, TResult>(
    T1 arg1,
    T2 arg2
)

public delegate TResult Func<T1, T2, T3, TResult>(
    T1 arg1,
    T2 arg2,
    T3 arg3
)

public delegate TResult Func<T1, T2, T3, T4, TResult>(
    T1 arg1,
    T2 arg2,
    T3 arg3,
    T4 arg4
)
```

Jokaisessa Func-delegaatin määrittelyssä geneerinen parametri TResult tarkoittaa delegaatin palautusarvon tyyppiä, ja se on aina parametrilistan viimeisenä, muut arvot, T1, T2, T3 ja T4, tarkoittavat metodin saamien parametrien tyyppejä.

Func-delegaatista on monta eri versiota, koska standardihakuoperaattorit saavat parametriksi delegaatteja, joilla on eri määrä parametreja. Yksikään standardihaku-operaattori ei saa parametrikseen delegaattia, joka vaatisi enemmän kuin neljä parametria.

Esimerkki 20: Yksi Where standardihakuoperaattorin prototyypeistä.

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate
)
```

Esimerkissä 20 on esitelty yksi Where-standardihakuoperaattorin prototyypeistä. Metodin parametrilistasta nähdään, että predicate niminen parametri on Func<TSource, bool> tyyppiä oleva delegaatti, eli operaattorille annettavan delegaatin pitää ottaa TSource tyyppiä oleva parametri ja palauttaa totuusarvo. Palautusarvon tyypin näkee siitä, että delegaatin viimeinen parametri on tyyppiä bool.

Näin ollen mille tahansa standardihakuoperaattorille, joka ottaa parametrikseen Func-tyyppisen delegaatin on helppo kirjoittaa delegaatti lambda-lausekkeena, kun tietää, miten se on määritelty standardihakuoperaattorin prototyypissä. Jos standardihakuoperaattorit ottaisivat parametrikseen erityyppisiä ei-geneerisiä delegaatteja, pitäisi ohjelmoijan muistaa ulkoa lukuisia erilaisia delegaattien määrittelyitä, jotta niitä voitaisiin käyttää standardihakuoperaattoreiden kanssa.

Func-delegaattia voi myös käyttää muun ohjelmoinnin yhteydessä kuin pelkästään standardihakuoperaattorien kanssa.

5 Viivästetyt standardihakuoperaattorit

Tässä luvussa esitellään viivästetyt standardihakuoperaattorit jaoteltuina käyttötarkoituksen mukaan.

Standardihakuoperaattoreiden esittelyyn tarvitaan tietolähde, jota vasten suoritetaan hakuja. Tämän tietolähteen lähdekoodi on listattuna liitteessä 1. ExampleData-luokka sisältää neljä staattista kokoelma ominaisuutta: Words, Numbers, Products ja Categories. Words-ominaisuus palauttaa staattisen jäsenmuuttujan words määrittelemän merkkijonotaulukon, ja Numbers palauttaa kokonaislukutaulukon sisältäen numerot 0-99, jotka generoidaan käyttämällä standardihakuoperaattoria Range.

Tietolähde sisältää myös kaksi toisiinsa liittyvää luokkaa, Productin ja Categoryn, jotka kuvaavat kategoriaa, johon kuuluu tuotteita. Vastaavat luokat voisivat kuulua esimerkiksi verkkokauppa tyyppiseen sovellukseen. Todellisuudessa luokilla olisi enemmänkin ominaisuuksia. Tuotteella on ominaisuutena kategorian id, johon tuote kuuluu, tällä id-arvolla voidaan liittää tuote kategoriaan. Products-ominaisuus palauttaa listan tuotteita ja Categories-ominaisuus palauttaa listan kategorioita.

Liitteessä 2 on koottu taulukkoon standardihakuoperaattorit, niiden palautusarvon tyyppi, käyttötarkoitus, onko operaattori viivästetty ja standardihakuoperaattoria vastaava hakusyntaksi, jos operaattori tukee sitä. Standardihakuoperaattorin palautusarvon tyypissä T tarkoittaa geneeristä tyyppiä, joka määräytyy standardihakuoperaattorin saamista parametreista. [1, s. 59–61; 8.]

5.1 Rajoitusoperaatiot

Rajoitusoperaatioita käytetään rajoittamaan hakuun sisältyviä elementtien määrää. Where-standardihakuoperaattori on ainut rajoitusoperaatio. Se suodattaa hakutulokseen sisältyvät elementit annetun ehtolausekkeen mukaisesti.

Where-operaattorille annetaan ehtolauseke Func<T, bool>-tyyppisenä predicate delegaattina, jossa T on sekvenssin tyyppi. Kun Where-operaattorin palauttamaa sekvenssiä käydään läpi, kutsutaan annettua delegaattia jokaisen elementin yhteydessä, jos delegaatti palauttaa totuusarvon tosi, niin elementti otetaan mukaan tulosjoukkoon.

Esimerkki 21: Where-operaattorin käyttö metodisyntaksilla ja hakusyntaksilla.

```
IEnumerable<int> result1 = ExampleData.Numbers
    .Where(n => n < 10);

foreach (int i in result1)
    Console.Write("{0} ", i);

Console.WriteLine();

// Hakusyntaksi
IEnumerable<int> result2 = from int i in ExampleData.Numbers
                           where i < 10
                           select i;

foreach (int i in result2)
    Console.Write("{0} ", i);
```

Esimerkissä 21 esitellään Where-standardihakuoperaattorin käyttö. Esimerkissä haetaan Numbers-taulukosta kaikki alle kymmenen olevat luvut. Esimerkissä on myös esitelty hakusyntaksin vastine Where-operaattorille.

Where-operaattorista on myös indeksoitu versio, jossa Func<T, int, bool>-tyyppiselle predicate-delegaatille annetaan sekvenssin elementin lisäksi elementin indeksiluku sekvenssissä. Indeksiluku alkaa aina nollasta.

5.2 Projektiot

Projektiolla tarkoitetaan operaatiota, jolla muutetaan olio uuteen muotoon, joka koostuu usein vain osasta vanhan olion ominaisuuksia. Projektio-operaatiot palauttavat sekvenssin uusia elementtejä, jotka on generoitu lähdesekvenssin elementeistä.

Select

Select-operaattori muuntaa saadun elementin uudeksi elementiksi. Se saa parametrikseen Func<TSource, TResult>-tyyppiä olevan selector-delegaatin, jonka tarkoitus on palauttaa lähdesekvenssin elementistä uusi muunnettu elementti.

Select-standardihakuoperaattorista on myös indeksoitu versio, jonka Func<TSource, int, TResult>-tyyppinen selector-delegaatti saa parametrikseen lähdesekvenssin elementin lisäksi elementin indeksiluvun lähdesekvenssissä.

Select-operaatiota vastaava hakusyntaksi on seuraava:

```
select [new object]
```

Esimerkki 22: Select-standardihakuoperaattorin käyttö.

```
// Metodisyntaksi
IEnumerable<int> result1 = ExampleData.Numbers
    .Where(n => n < 10)
    .Select(n => n * 2);

// Hakusyntaksi
IEnumerable<int> result2 = from n in ExampleData.Numbers
                           where n < 10
                           select n * 2;

foreach (int i in result1)
    Console.Write("{0} ", i);

Console.WriteLine();

foreach (int i in result2)
    Console.Write("{0} ", i);
```

Esimerkissä 22 on esitelty Select-operaattorin käyttö metodisyntaksilla ja hakusyntaksilla. Esimerkissä haetaan Numbers-taulukosta alle kymmenen olevat luvut, joille suoritetaan projektio, jossa luku kerrotaan kahdella.

Kun Select-operaatiolla muodostetaan uusi olio, jonka tyyppiä ei ole etukäteen määritelty, ei hakua voida sijoittaa vahvasti tyypitettyyn sekvenssiin vaan on käytettävä var-muuttujaa.

Esimerkki 23: Select-standardihakuoperaation indeksoidun version käyttö.

```
var result = ExampleData.Categories
    .Select( (c, indx) => new { Index = indx, Name = c.Name } );

foreach (var c in result)
    Console.WriteLine("Index: {0} Name: {1}", c.Index, c.Name);
```

Esimerkissä 23 esitellään Select-operaattorin indeksoidun version käyttö. Esimerkissä haetaan kaikki kategoriat, joista projisoidaan uusi anonyymi tyyppi, joka sisältää elementin indeksiluvun ja kategorian nimen.

SelectMany

SelectMany-operaattoria käytetään 1-N-projektioon. SelectMany palauttaa jokaista lähdesekvenssin elementtiä kohden uuden sekvenssin, joka koostuu nollasta tai useammasta elementistä ja yhdistää nämä sekvenssit yhdeksi sekvenssiksi.

SelectMany-standardihakuoperaattori saa parametrikseen Func<TSource, IEnumerable<TResult>>-tyyppisen selector-delegaatin, jonka tarkoituksena on palauttaa sekvenssi elementtejä yhtä lähdesekvenssin elementtiä kohden. Operaattorille voidaan myös antaa lisäksi Func<TSource, TCollection, TResult>-tyyppinen resultSelector-delegaatti, joka on tarkoitettu palautettavan sekvenssin elementtien jatkokäsittelyyn. SelectMany-standardihakuoperaattorista on myös indeksoitu versio.

Esimerkki 24: SelectMany-standardihakuoperaattorin käyttö.

```
var productsByCategory = ExampleData.Categories
    .SelectMany(
        cat => ExampleData.Products
            .Where(prod => prod.CategoryId == cat.Id),
        (cat, prod) =>
            new {CategoryName=cat.Name, ProductName=prod.Name}
     );

foreach (var p in productsByCategory)
    Console.WriteLine("{0} - {1}", p.CategoryName, p.ProductName);
```

Esimerkissä 24 haetaan SelectMany-operaattorilla kaikkien tuotteiden nimi ja vastaavan tuotteen kategorian nimi. Esimerkissä haetaan ensiksi kategoriat ja niihin kuuluvat tuotteet selector-delegaatissa ja muodostetaan resultSelector-delegaatissa lopputuloksesta uusi anonyymi olio, jolle annetaan ominaisuuksiksi kategorian ja tuotteen nimi. Esimerkin tulossekvenssin läpikäyntiin riittää vain yksi luuppi, koska SelectMany-operaattori yhdistää palautettavat sekvenssit yhdeksi sekvenssiksi.

5.3 Ositusoperaatiot

Ositusoperaatioilla voidaan jakaa lähdesekvenssi kahteen osajoukkoon ja palauttaa toinen niistä.

Take

Take-operaattori palauttaa annetun lukumäärän verran elementtejä sekvenssin alusta. Se saa parametrikseen kokonaisluvun, jonka verran elementtejä sen pitää palauttaa.

Esimerkki 25: Take-standardihakuoperaattorin käyttö.

```
IEnumerable<string> result = ExampleData.Words.Take(5);

foreach (string s in result)
    Console.Write(s + " ");
```

Esimerkissä 25 haetaan viisi ensimmäistä sanaa Words-taulukosta käyttämällä Take-operaattoria.

TakeWhile

TakeWhile-operaattori palauttaa elementtejä lähdesekvenssistä niin kauan kuin sille annettu ehto täyttyy, jos esimerkiksi sekvenssin ensimmäinen elementti ei täytä annettua ehtoa, palautettaan tyhjä sekvenssi. TakeWhile-operaattori saa ehtolausekkeen Func<TSource, bool>-tyyppisenä predicate-delegaattina. Operaattorista on myös indeksoitu versio, jonka predicate-delegaatti on tyyppiä Func<TSource, int, bool>.

Esimerkki 26: TakeWhile-standardihakuoperaattorin käyttö.

```
IEnumerable<string> result = ExampleData.Words
    .OrderBy( s => s.Length)
    .TakeWhile(s => s.Length < 4);

foreach (string s in result)
    Console.WriteLine(s);
```

Esimerkissä 26 ensiksi järjestetään Word-taulukon sanat pituuden mukaan nousevassa järjestyksessä ja sitten haetaan sanoja niin kaunan, kuin sanan pituus on alle neljä merkkiä.

Skip

Skip-standardihakuoperaattori ohittaa annetun lukumäärän verran elementtejä lähdesekvenssistä. Jos ohitettavien elementtien lukumäärä on suurempi kuin elementtien määrä lähdesekvenssissä, Skip-operaattori palauttaa tyhjän sekvenssi.

Esimerkki 27: Skip-standardihakuoperaattorin käyttö.

```
IEnumerable<int> result = ExampleData.Numbers.Skip(80);

foreach (int n in result)
    Console.Write("{0} ", n);
```

Esimerkissä 27 haetaan Numbers-taulukon luvut ja ohitetaan 80 ensimmäistä numeroa.

SkipWhile

SkipWhile-standardihakuoperaattori ohittaa lähdesekvenssin elementtejä niin kauan kuin sille annettu ehto on tosi ja palauttaa jäljelle jäävät elementit.

SkipWhile operaattori saa ehtolausekkeen Func<TSource, bool>-tyyppisenä predicate-delegaattina, jonka pitää palauttaa totuusarvo siitä, ohitetaanko kyseinen lähdesekvenssin elementti. SkipWhile-standardioperaattorista on myös indeksoitu versio, jonka predicate-delegaatti on tyyppiä Func<TSource, int, bool>.

Esimerkki 28: SkipWhile-standardihakuoperaattorin käyttö.

```
IEnumerable<string> result = ExampleData.Words
    .OrderBy( s => s.Length )
    .SkipWhile( s => s.Length <= 8 );

foreach (string s in result)
    Console.Write("{0} ", s);
```

Esimerkissä 28 järjestetään ensiksi Words-taulukko sanojen pituuden mukaan nousevaan järjestykseen, jonka jälkeen ohitetaan sanoja niin kauan kuin sanan pituus on kahdeksan tai alle kahdeksan merkkiä.

5.4 Järjestysoperaatiot

Järjestysoperaatioilla voidaan järjestää lähdesekvenssi nousevaan tai laskevaan järjestykseen järjestysavaimen mukaan. Järjestysoperaattorit ottavat parametrikseen IEnumerable<T>-tyyppisen sekvenssin ja palauttavat IOrderedEnumerable<T>-tyyppisen ns. järjestetyn sekvenssin, joten esimerkiksi OrderBy-järjestysoperaattorin palauttamalle sekvenssille ei voida enää kutsua uudelleen OrderBy-operaattoria. Jos jo kertaalleen järjestetyn sekvenssin järjestystä halutaan tarkentaa eli järjestää sekvenssissä samalla järjestysavaimella olevat elementit jonkin muun järjestysavaimen mukaan, pitää käyttää ThenBy- tai ThenByDescending-järjestysoperaattoreita.

Järjestysoperaattorit suorittavat stabiilin järjestämisen, eli jos lähdesekvenssissä on samalla järjestysavaimen arvolla useampi elementti, niin järjestysoperaattorit eivät muuta näiden elementtien alkuperäistä järjestystä vaan elementit ovat tulossekvenssissä samassa järjestyksessä kuin ne esiintyvät lähdesekvenssissä.

Järjestysoperaattorit suorittavat järjestämisen käyttämällä järjestysavaimen oletuskomparaattoria, eli IComparable<T>-rajapinnan Compare-metodia, tai lisäparametrina annettavaa IComparer<T>-rajapinnan toteuttavan komparaattoriolion Compare-metodia.

OrderBy ja OrderByDescending

OrderBy- ja OrderByDescending-järjestysoperaattorit ovat muuten identtisiä, erona on vain se, että OrderBy-operaattori järjestä lähdesekvenssin nousevaan järjestykseen ja OrderByDescending järjestää lähdesekvenssin laskevaan järjestykseen.

Molemmat operaattorit ottavat parametrikseen Func<TSource, TKey>-tyyppisen keySelector-delegaatin, jonka tarkoitus on palauttaa lähde-elementistä järjestysavain, jonka mukaan järjestäminen tapahtuu.

OrderBy-operaattorin hakusyntaksi on muotoa:

```
orderby [TKey]
```

OrderByDescending-operaattorin hakusyntaksi on sama kuin OrderBy-operaattorin, mutta järjestysavaimen perään lisätään vain määritys laskevasta järjestyksestä:

```
orderby [TKey] descending
```

Esimerkki 29: OrderBy- ja OrderByDescending-standardihakuoperaattorien käyttö.

```
// Metodisyntaksi
IEnumerable<string> res1 = ExampleData.Words.OrderBy(w => w);

// Hakusyntaksilla
IEnumerable<string> res2 = from w in ExampleData.Words
                           orderby w descending
                           select w;

foreach (string s in res1.Take(10))
    Console.WriteLine(s);

Console.Write("---{0}", Environment.NewLine);

foreach (string s in res2.Take(10))
    Console.WriteLine(s);
```

Esimerkissä 29 on esitelty OrderBy- ja OrderByDescending-operaattorien käyttö. Esimerkissä järjestetään kaikki Words-taulukon sanat nousevaan ja laskevaan järjestykseen hakusyntaksilla ja metodisyntaksilla. Lopuksi tulostetaan ensimmäiset kymmenen sanaa.

ThenBy ja ThenByDescending

ThenBy- ja ThenByDescending-standardihakuoperaattoreita käytetään järjestämään jo järjestyksessä oleva IOrderedEnumerable<T>-sekvenssi uudelleen toisen järjestysavaimen mukaan. Operaattorit eroavat toisistaan vain sillä, että ThenBy järjestää sekvenssin nousevaan järjestykseen ja ThenByDescending järjestää sekvenssin laskevaan järjestykseen.

ThenBy- ja ThenByDescending-operaattorit toimivat samalla tavoin kuin OrderBy- ja OrderByDescending-standardihakuoperaattorit, eli ne saavat parametrikseen Func<TSource, TKey>-tyyppisen keySelector-delegaatin, jonka pitää palauttaa järjestysavain, jonka mukaan järjestäminen suoritetaan.

ThenBy-operaattorin hakusyntaksi on sama kuin OrderByn ja seuraavat ThenBy-operaattorin saamat järjestysavaimet erotellaan pilkulla toisistaan:

```
orderby [Tkey1], [TKey2], ...
```

ThenByDescending-operaattorin hakusyntaksi on sama kuin ThenByn, mutta järjestysavaimen perään lisätään descending-määre:

```
orderby [Tkey1], [Tkey2] descending, ...
```

Esimerkissä 30 esitellään ThenBy- ja ThenByDescending-standardihakuoperaattoreiden käyttö. Esimerkissä järjestetään Products-listan tuotteet ensiksi OrderBy-operaattorilla tuotteen kategorian id:n mukaan ja sitten ThenBy-operaattorilla tuotteen id:n mukaan. Hakusyntaksilla tehdään sama haku kuin metodisyntaksilla, mutta tuotteiden id:n mukainen

järjestys tehdään laskevassa järjestyksessä. Lopuksi tulostetaan tuotteen kategorian id, tuotteen nimi ja suluissa tuotteen id.

Esimerkki 30: ThenBy- ja ThenByDescending-standardihakuoperaattorien käyttö.

```
// Metodisyntaksi
IEnumerable<Product> res1 = ExampleData.Products
    .OrderBy( p => p.CategoryId )
    .ThenBy( p => p.Id);

// Hakusyntaksilla
IEnumerable<Product> res2 = from p in ExampleData.Products
                           orderby p.CategoryId, p.Id descending
                           select p;

foreach (Product p in res1)
    Console.WriteLine("{0} - {1} ({2})", p.CategoryId,
        p.Name, p.Id);

Console.Write("---{0}", Environment.NewLine);

foreach (Product p in res2)
    Console.WriteLine("{0} - {1} ({2})", p.CategoryId,
        p.Name, p.Id);
```

Reverse

Reverse-operaattori ei suorita oikeaa järjestämistä, vaan se palauttaa lähdesekvenssin elementit vastakkaisessa järjestyksessä.

Esimerkki 31: Reverse-standardihakuoperaattorin käyttö.

```
int[] arr = {0, 1, 2, 3, 2, 5};

foreach (int i in arr.Reverse())
    Console.WriteLine(i);
```

Esimerkissä 31 arr-kokonaislukutaulukon sisältö tulostetaan käänteisessä järjestyksessä.

5.5 Joukko-operaatiot

Joukko-operaatiolla voidaan suorittaa lähdesekvenssille matemaattisia joukko-operaatioita, kuten osittamista tai leikkauksia. Joukko-operaatiot perustuvat sekvenssin tai sekvenssien elementtien yhdenvertaisuuteen. Elementtien yhdenvertaisuuden tarkistukseen käytetään oletusvertailumetodeja, eli Object-luokan GetHashCode- ja Equals-metodeja tai joukko-operaattorille erikseen parametrina annettavaa IEqualityComparer<T>-rajapinnan toteuttavaa vertailuoliota.

Distinct

Distinct-standardihakuoperaattori palauttaa sekvenssistä vain toisistaan eroavat elementit.

Esimerkki 32: Distinct-standardihakuoperaattorin käyttö.

```
int[] array = { 0, 2, 1, 2, 6, 5, 6 };

foreach (int i in array.Distinct())
    Console.WriteLine(i);
```

Esimerkissä 32 luodaan kokonaislukutaulukko, joka sisältää saman lukuarvon useampaan kertaan. Tästä kokonaislukutaulukosta haetaan Distinct-operaattorilla toisistaan eroavat luvut. Lukujen 2 ja 6 duplikaatit eivät esiinny hakutuloksesta.

Except

Except-operaattori palauttaa lähdesekvenssin josta on poistettu parametrina saadun sekvenssin yhdenvertaiset elementit.

Esimerkki 33: Except-standardihakuoperaattorin käyttö.

```
int[] numbers = { 0, 1, 2, 3, 4, 5 };
int[] not = { 0, 3, 5 };

IEnumerable<int> result = numbers.Except(not);

foreach (int i in result)
    Console.WriteLine(i);
```

Esimerkissä 33 luodaan kaksi kokonaislukutaulukkoa: numbers ja not. Numbers-taulukosta haetaan Except-operaattorilla ne luvut, jotka eivät ole not-taulukossa.

Intersect

Intersect-operaattori suorittaa leikkausoperaation kahden sekvenssin välillä, eli se palauttaa sekvenssin elementtejä, jotka esiintyvät sekä lähdesekvenssissä että parametrina saadussa sekvenssissä.

Esimerkki 34: Intersect-standardihakuoperaattorin käyttö.

```
int[] first = { 0, 1, 2, 3, 4, 5 };
int[] second = { 9, 3, 5 };

IEnumerable<int> result = first.Intersect(second);

foreach (int i in result)
    Console.WriteLine(i);
```

Esimerkissä 34 tehdään kaksi kokonaislukutaulukkoa, first ja second, ja tulostetaan ne luvut jotka esiintyvät molemmissa taulukoissa.

Union

Union-operaattori suorittaa yhdistysoperaation lähdesekvenssin ja parametrina saadun sekvenssin välillä, eli Union-operaattorin palauttama sekvenssi sisältää ne elementit, jotka

kuuluvat jompaankumpaan tai molempiin sekvensseihin. Union-operaattori poistaa duplikaatit elementit palautettavasta sekvenssistä.

Esimerkki 35: Union-standardihakuoperaattorin käyttö.

```
int[] first = { 1, 2, 3, 4, 5 };
int[] second = { 9, 3, 5, 0, 2 };

IEnumerable<int> result = first.Union(second);

foreach (int i in result)
    Console.Write("{0} ", i );
```

Esimerkissä 35 käytetään Union-operaattoria yhdistämään first- ja second-kokonaislukutaulukot.

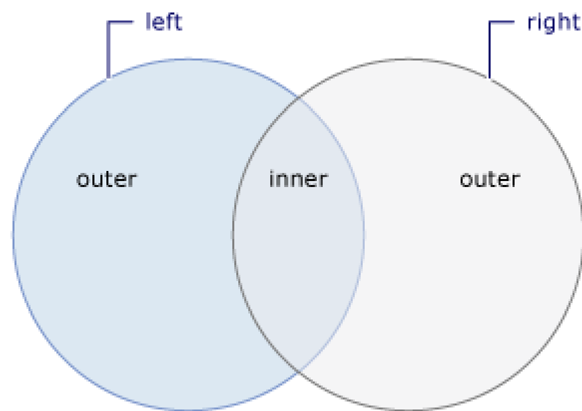
5.6 Liitosoperaatiot

Liitosoperaatiolla voidaan yhdistää useita sekvenssejä toisiinsa sekvenssien elementtien yhteisen avaimen perusteella. Liitosavain voi koostua useasta elementin ominaisuudesta, jolloin puhutaan ns. komposiittiavaimesta.

Liitosoperaatiot ovat yksi tärkeimmistä standardihakuoperaattoreista silloin kun hakujen kohteena on tietolähde, jossa olioiden relaatioita toisiinsa ei voida suoraan seurata. Esimerkkitietolähteen Category- ja Product-luokat liittyvät toisiinsa epäsuorasti, eli esimerkiksi Category-olio ei sisällä seurattavaa relaatiota siihen liittyviin Product-olioihin, joten Category-oliosta ei voida suoraan hakea sen sisältämiä Product-olioita. Tähän tarvitaan liitosoperaatiota.

Join- ja GroupJoin-operaattorit suorittavat yhdenvertaisuusliitoksen, joka perustuu toisiinsa liitettävien tietolähteiden yhdenvertaisiin avaimiin.

Join-standardihakuoperaattori suorittama liitos vastaa relaatiotietokantojen sisäliitosta, jossa hakutulokseen saadaan ne elementit, joille löytyy vastaava elementti liitettävästä sekvenssistä. GroupJoin-operaattorille ei ole vastaavaa relaatiotietokannan liitosta, mutta sen toiminta vastaa joukkoa sisäliitoksia ja ulkoliitoksia. Kuvassa 3 havainnollistetaan liitostyyppejä.



Kuva 3: Liitostyypit. [9.]

Sisäliitoksella tarkoitetaan kuvan left-sekvenssin liittämistä kuvan right-sekvenssin kanssa, siten että lopputulokseksi saadaan kuvan ympyröiden leikkaava alue (inner).

Ulkoliitoksella (left outer join) tarkoitetaan sitä, että liitoksessa saadaan tulosjoukkoon kuvan left-sekvenssin kaikki elementit, vaikka niillä ei olisi vastinetta right-sekvenssissä.

Join- ja GroupJoin-liitosoperaattorit käyttävät liitokset avainten vertailuun oletusvertailuoperaattoreita tai lisäparametrina annettavaa IEqualityComparer<T>-rajapinnan toteuttavaa vertailuolioita.

Join

Join-operaattori suorittaa sisäliitoksen, jossa liitetään kaksi sekvenssiä toisiinsa perustuen elementtien yhtäläisiin avaimiin, jotka poimitaan molemmista sekvensseistä. Join-operaattori on tarkoitettu liittämään 1-1-relaatiossa olevat sekvenssit toisiinsa.

Join-standardihakuoperaattorille annetaan liitettävän inner-sekvenssin lisäksi Func<TOuter, Tkey>-tyyppinen outerKeySelector-delegaatti ja Func<TInner, TKey>-tyyppinen innerKeySelector-delegaatti, joiden tarkoitus on palauttaa inner- ja outer-sekvenssien elementeistä liitokseen tarvittavat avaimet. Näiden avaimien yhdenvertaisuuden perusteella päätellään kuuluko outer- ja inner-sekvenssien kyseiset elementit tulosjoukkoon. Jos elementit kuuluvat tulosjoukkoon, eli niiden avaimet vastaavat toisiaan, annetaan molemmat elementit Func<TOuter, TInner, TResult>-tyyppiselle resultSelector-delegaatille, jonka tarkoitus on muodostaa näistä elementeistä projektio.

Join-operaattorin hakusyntaksi on muotoa:

```
from [objectA] in [outerDataSource]
join [objectB] in [innerDataSource] on [outerKey] equals [innerKey]
```

Join-operaattorin hakusyntaksissa annetaan liitettävä sekvenssi kuten from lausekkeessa, mutta käytetään ”join” määrettä. Liitosavaimet määritetään ”on” määreen jälkeen siten, että ”equals” määreen vasemmalle puolelle tulee ulomman liitoksen avain ja oikealle puolelle liitettävän sekvenssin avain. Ks. Kuva 3.

Esimerkki 36: Join-standardihakuoperaattorin käyttö.

```
// Metodisyntaksi
var result1 = ExampleData.Products
    .Where( p => p.Id == 1)
    .Join( ExampleData.Categories,
        prod => prod.CategoryId,
        cat => cat.Id,
        (prod, cat) => new {CategoryName = cat.Name,
                ProductName = prod.Name });

// Hakusyntaksi
var result2 = from prod in ExampleData.Products
              join cat in ExampleData.Categories
                on prod.CategoryId equals cat.Id
              where prod.Id == 1
              select new {CategoryName = cat.Name,
                ProductName = prod.Name };

foreach (var prod in result1)
    Console.WriteLine("{0} - {1}", prod.CategoryName,
        prod.ProductName);

foreach(var prod in result2)
    Console.WriteLine("{0} - {1}", prod.CategoryName,
        prod.ProductName);
```

Esimerkissä 36 esitellään Join standardihakuoperaattorin käyttö. Esimerkissä haetaan Products-tuotelistasta tuote, jonka id-luku on yksi ja liitetään tähän tuotteeseen tuotteen kategoria Categories-listasta, käyttämällä liitosavainparina Product.CategoryId- ja Category.Id-ominaisuuksia. Lopuksi muodostetaan anonyymi olio, jonka ominaisuuksiksi asetetaan tuotteen nimi ja kategorian nimi. Esimerkissä tehdään sama haku metodisyntaksilla ja hakusyntaksilla.

GroupJoin

GroupJoin-operaattorilla voidaan suorittaa sisä- tai ulkoliitos, jossa liitetään kaksi sekvenssiä toisiinsa perustuen elementtien yhtäläisiin avaimiin. GroupJoin-operaattoria käytetään 1-N-relaatiossa olevien olioiden liittämiseen toisiinsa.

GroupJoin-operaattori toimii samalla tavoin kuin Join-operaattori sillä poikkeuksella, että se palauttaa yhtä outer-sekvenssin elementtiä kohden sekvenssin inner-sekvenssin elementtejä, siinä missä Join palautti yhden inner-sekvenssin elementin outer-sekvenssin elementtiä kohden.

GroupJoin-operaattorin parametrit vastaavat Join-operaattoria, mutta sen Func<TOuter, IEnumerable<TInner>, Tresult>-tyyppinen resultSelector-delegaatti saa nyt yhden sekvenssin elementtejä, jokaista outer-sekvenssin elementtiä kohden. Jos inner-sekvenssistä ei löydy vastaavia elementtejä, on resultSelector-delegaatin saama IEnumerable<TInner>-tyyppiä olevan parametrin arvo null, eli resultSelector-delegaattia kutsutaan kaikille outer-sekvenssin elementeille, siitä huolimatta, löytyikö niille vastinetta inner-sekvenssistä.

GroupJoin-operaattorin hakusyntaksi vastaa Join-operaattorin hakusyntaksia, mutta join lausekkeen loppuun sijoitetaan ”into” määre, jolla sijoitetaan liitoksen tuottama sekvenssi muuttujaan jatkokäsittelyä varten.

```
from [objectA] in [outerDataSource]
join [objectB] in [innerDataSource]
    on [outerKey] equals [innerKey] into [groupB]
```

Esimerkki 37: GroupJoin-standardihakuoperaattorin käyttö.

```
var result1 = ExampleData.Categories
    .Where(cat => cat.Id == 1)
    .GroupJoin(ExampleData.Products,
        cat => cat.Id,
        prod => prod.CategoryId,
        (cat, products) => new { cat.Name, Products = products });


var result2 = from cat in ExampleData.Categories
              where cat.Id == 2
              join prod in ExampleData.Products
                on cat.Id equals prod.CategoryId into products
              select new { cat.Name, Products = products };

foreach (var cat in result1.Concat(result2) )
{
    Console.WriteLine(cat.Name);
    foreach (Product p in cat.Products)
        Console.WriteLine("    " + p.Name);
}
```

Esimerkissä 37 esitellään GroupJoin-operaattorin käyttö. Esimerkissä haetaan kategoria ja sen kaikki tuotteet. Tämä haku tehdään ensiksi metodisyntaksilla kategorialle id luvulla 1 ja hakusyntaksilla kategorialle id luvulla 2. Ennen tulostusta sekvenssit yhdistetään toisiinsa Concat-operaattorilla, jotta molemmat tulokset voidaan käsitellä samassa luupissa. Esimerkin haussa käytetään kategorian ja tuotteiden liitosavainparina Products.CategoryId- ja Category.Id-ominaisuuksia.

Esimerkistä nähdään että GroupJoin suorittaa 1-N liitoksen, joka näkyy siinä, että kun luodaan hakutuloksesta anonyymia oliota resultSelector delegaatissa, sijoitetaan anonyymin olion Products ominaisuuteen GroupJoin tuottama products sekvenssi. Tämä sama näkyy myös siinä, että tulostettaessa kategorian nimeä ja sen tuotteiden nimiä, joudutaan käyttämään kahta sisäkkäistä luuppia.

GroupBy-operaattorilla voidaan tehdä sisäliitos tarkistamalla haun yhteydessä sisältääkö resultSelector-delegaatin saama IEnumerable<TInner>-tyyppinen sekvenssi elementtejä, esimerkiksi käyttäen Any-standardihakuoperaattoria. Esimerkissä 38 tehdään sama haku kuin edellisessä esimerkissä, sillä erona että kategoriat tallennetaan aluksi listaan johon lisätään vielä yksi kategoria jolla ei ole tuotteita. Tästä listasta sitten haetaan vain ne kategoriat, joilla on myös tuotteita.

Esimerkki 38: Sisäliitoksen teko GroupBy standardihakuoperaattorilla.

```
List<Category> categories = ExampleData.Categories;
// Lisätään uusi kategoria ilman tuotteita
categories.Add(new Category { Id = 99, Name = "Temp" });

var result = from cat in categories
             join prod in ExampleData.Products
                on cat.Id equals prod.CategoryId into products
             where products.Any()
             select new { cat.Name, Products = products };

foreach (var cat in result)
{
    Console.WriteLine(cat.Name);
    foreach (Product p in cat.Products)
        Console.WriteLine("    " + p.Name);
}
```

5.7 Ryhmittelyoperaatiot

Ryhmittelyoperaatioilla voidaan järjestää sekvenssin elementit ryhmiin, jotka jakavat yhden tai useamman yhteisen ominaisuuden.

GroupBy

GroupBy-operaattoria käytetään lähdesekvenssin ryhmittämiseen elementtien yhteisen avaimen perusteella. Osa GroupBy-operaattorin versioista palauttavat sekvenssin IGrouping<TKey, T>-tyyppisiä elementtejä, eli sekvenssin ryhmiä.

IGrouping<TKey, T>-rajapinta esittää ryhmää T-tyyppisiä elementtejä TKey-tyyppiä olevalla avaimella. Rajapinta periytyy IEnumerable<T>-rajapinnasta ja sillä on Key-ominaisuus, joka on ryhmän avain.

GroupBy käyttää ryhmittelyavainten vertailussa oletusvertailuoperaattoria tai erikseen parametrilla annettavaa IEqualityComparer<T>-rajapinnan toteuttavaa vertailuoliota.

GroupBy-operaattori saa parametrina Func<TSource, Tkey>-tyyppisen keySelector-delegaatin, jonka tarkoitus on palauttaa lähdesekvenssin elementeistä ryhmittelyyn käytettävä avain, tämän avaimen perusteella lähdesekvenssin elementeistä muodostetaan IGrouping<TKey, TSource>-tyyppinen sekvenssi ryhmiä.

Esimerkki 39: Sekvenssin ryhmittely GroupBy operaattorilla pelkän avaimen mukaan.

```
IEnumerable<IGrouping<int, Product>> productGroups =
    ExampleData.Products.GroupBy(p => p.CategoryId);

foreach (IGrouping<int, Product> group in productGroups)
{
    Console.WriteLine("GroupKey: {0}", group.Key);
    foreach (Product p in group)
        Console.WriteLine("   {0}", p.Name);
}
```

Esimerkissä 39 on esitelty sekvenssin ryhmittely GroupBy-operaattorilla pelkän avaimen mukaan. Esimerkissä haetaan kaikki tuotteet ja ryhmitellään ne tuotteen CategoryId-ominaisuuden mukaan ja tulostetaan ryhmien avain ja ryhmien tuotteiden nimet.

GroupBy-operaattorille voidaan antaa lisäparametrina Func<TSource, TElement>-tyyppinen elementSelector-delegaatti, joka suorittaa projektion ryhmään valituille elementeille.

Esimerkki 40: Sekvenssin ryhmittely GroupBy-operaattorilla ja elementtikohtaisen projektion suorittaminen.

```
IEnumerable<IGrouping<int, string>> productNameGroups =
    ExampleData.Products.GroupBy(p => p.CategoryId, p => p.Name );

foreach (IGrouping<int, string> group in productNameGroups)
{
    Console.WriteLine("GroupKey: {0}", group.Key);
    foreach (string productName in group)
        Console.WriteLine("   {0}", productName);
}
```

Esimerkissä 40 tehdään sama ryhmittely kuin edellisessä esimerkissä, mutta nyt projisoidaan tulosjoukkoon pelkästään tuotteen nimi elementSelector-delegaatissa.

GroupBy-operaattorilla voidaan suorittaa projektio koko ryhmälle, antamalla sille lisäparametrina Func<TKey, IEnumerable<TSource>, TResult>-tyyppinen resultSelector-delegaatti, joka saa parametrikseen ryhmän avaimen ja ryhmän elementtisekvenssin. Tällöin GroupBy-operaattori palauttaa normaalin IEnumerable<T>-tyyppisen sekvenssin, jonka resultSelector-delegaatti on projisoinut.

Esimerkki 41: Sekvenssin ryhmittely ja projisointi GroupBy-operaattorilla.

```
int[] numbers = {0, 1, 2, 3, 4, 5, 6, 7, 8};

var result = numbers
    .GroupBy(
        n => (n % 2 == 0) ? "Even numbers: " : "Odd numbers: ",
        (key, nums) => new {Key = key, Count = nums.Count() }
    );

foreach (var r in result)
    Console.WriteLine("{0} : {1}", r.Key, r.Count);
```

Esimerkissä 41 ryhmitellään numbers-taulukon luvut parillisiin ja parittomiin lukuihin GroupBy-operaattorilla, ryhmien avaimena toimivat merkkijonot ”Even numbers:” ja ”Odd numbers: ”. Operaattorin resultSelector-delegaatissa projisoidaan tulosryhmästä anonyymi olio, jolle valitaan ominaisuuksiksi ryhmän avain ja elementtien lukumäärä ryhmässä.

GroupBy-operaattorille voidaan myös suorittaa elementtikohtainen projektio ja ryhmän projektio antamalla sille molemmat, elementSelector- ja resultSelector-delegaatit.

GroupBy-operaattorin hakusyntaksista on kaksi muotoa. Ensimmäisellä voidaan pelkästään ryhmitellä hakutulos avaimen mukaan.

```
from [object] in [DataSource]
group [object] by [key]
```

Jos halutaan tehdä ryhmälle jälkikäsittelyä, voidaan ryhmä sijoittaa hakulausekkeessa apumuuttujaan ”into” määreellä, jolloin haku jatkuu normaalista ja lopuksi lopettaa haku uudella groupby- tai select-lausekkeella.

```
from [object] in [DataSource]
group [object] by [key] into [group]
...
```

Esimerkki 42: GroupBy-standardihakuoperaattorin hakusyntaksin käyttö.

```
var result =
    from product in ExampleData.Products
    group product by product.CategoryId into productGroups
    select new
    {
        CategoryId = productGroups.Key,
        ProductCount = productGroups.Count()
    };

foreach (var item in result)
{
    Console.WriteLine("Category {0} contains {1} products.",
        item.CategoryId, item.ProductCount);
}
```

Esimerkissä 42 esitellään GroupBy-standardihakuoperaattorin hakusyntaksin käyttö. Esimerkissä ryhmitellään tuotteet niiden kategorian id luvun mukaisiin ryhmiin ja tehdään projektio, jossa haetaan ryhmän avain ja ryhmän elementtien lukumäärä uuteen anonyymiin olioon. Koska tarvitaan ryhmän jälkikäsittelyä, esimerkissä sijoitetaan ryhmät ”into” määreellä productGroups-nimiseen apumuuttujaan.

5.8 Generointioperaatiot

Generointioperaatiot on tarkoitettu helpottamaan sekvenssien luontia. Kaikki generointioperaattorit paitsi DefaultIfEmpty-operaattori, on toteutettu Enumerable luokan staattisina metodeina.

Empty

Empty-standardihakuoperaattori palauttaa tyhjän IEnumerable<T>-tyyppisen sekvenssin.

DefaultIfEmpty

DefaultIfEmpty-operaattori palauttaa sekvenssin, joka sisältää oletusarvon tai lisäparametrina annetun elementin, jos lähdesekvenssi on tyhjä. Oletusarvo on referenssi- ja Nullable-tyypeille null ja arvotyypeille tyypin oletusarvo.

Esimerkki 43: DefaultIfEmpty-standardihakuoperaattorin käyttö.

```
IEnumerable<string> data = Enumerable.Empty<string>();

foreach (string s in data.DefaultIfEmpty("Empty sequence"))
    Console.WriteLine(s);
```

Esimerkissä 43 luodaan tyhjä sekvenssi Empty-standardihakuoperaattorilla, josta haetaan DefaultIfEmpty-standardihakuoperaattorilla ”Empty sequence” merkkijono, jos sekvenssi on tyhjä.

Range

Range-operaattorilla voidaan generoida ennalta määrätty kokonaislukusekvenssi. Standardihakuoperaattori palauttaa IEnumerable<int>-tyyppisen sekvenssin kokonaislukuja, jotka alkavat start-parametrilla annetusta alkuarvosta ja jatkuvat count-parametrin verran.

Esimerkki 44: Range-standardihakuoperaattorin käyttö.

```
IEnumerable<int> numbers = Enumerable.Range(1, 10);
foreach (int n in numbers)
    Console.Write("{0} ", n);
```

Esimerkissä 44 generoidaan Range-operaattorilla kokonaislukusekvenssi, joka sisältää kokonaisluvut 1–10.

Repeat

Repeat-operaattori generoi sekvenssin, joka sisältää yhden toistetun arvon. Metodi saa parametrikseen TResult-tyyppisen elementin, jonka se toistaa count-parametrina annetun määrän verran.

Esimerkki 45: Repeat-standardihakuoperaattorin käyttö.

```
Category cat = new Category{ Id=1, Name="Temp" };

IEnumerable<Category> cats = Enumerable.Repeat(cat, 3);

foreach (Category c in cats)
    Console.WriteLine("{0} - {1}", c.Id, c.Name);
```

Esimerkissä 45 luodaan ensiksi kategoria olio, joka toistetaan Repeat-operaattorilla kolmeen kertaan.

5.9 Muunnosoperaatiot

Muunnosoperaatioilla voidaan suorittaa lähdesekvenssille tai lähdesekvenssin elementeille tyyppimuunnos. Cast- ja OfType-standardihakuoperaattorit mahdollistavat standardihakuoperaattoreiden käytön ei-geneeristen IEnumerable-rajapinnan toteuttavien kokoelmien kanssa.

AsEnumerable

AsEnumerable-operaattori palauttaa lähdesekvenssin IEnumerable<T>-tyyppinä. Operaattorilla ei ole muuta vaikutusta kuin käännöksen aikaisen IEnumerable<T>-rajapinnan toteuttavan olion muuntaminen itse IEnumerable<T>-tyypiksi.

AsEnumerable-operaattori on hyödyllinen silloin kun haun kohteena oleva luokka sisältää jo vastaavia standardihakuoperaattoreiden omia metoditoteutuksia. Tällöin ei voida kutsua standardihakuoperaattoria, joka on toteutettu extension-metodina, koska luokan oma vastaava metodi on tällöin käännöksenaikaisen metodikutsun selvityksessä etusijalla, jolloin standardihakuoperaattoria ei koskaan tulla kutsumaan. AsEnumerable-operaattorilla voidaan piilottaa luokan omat metodit, jolloin saadaan standardihakuoperaattorien vastaavat metodit käyttöön.

Cast

Cast-operaattori tyyppimuuntaa IEnumerable-rajapinnan toteuttavan sekvenssin elementit annetuksi tyypiksi, ja palauttaa tyypitetyn IEnumerable<T> sekvenssin.

Jos jonkin lähdesekvenssin elementin tyyppimuunnos epäonnistuu, Cast-operaattori aiheuttaa InvalidCastException-poikkeuksen.

Cast-standardihakuoperaattorin hakusyntaksi on seuraava:

```
from [Type] [object] in [DataSource]
```

Cast-standardihakuoperaattorin hakusyntaksissa ilmaistaan eksplisiittisesti haun kohteena olevan olion tyyppi.

Esimerkki 46: Cast-standardihakuoperaattorin käyttö.

```
ArrayList oldArray = new ArrayList();
oldArray.Add("Hello ");
oldArray.Add("asd");
oldArray.Add("Linq ");
oldArray.Add("");
oldArray.Add("world!");

IEnumerable<string> result1 = oldArray.Cast<string>()
    .Where(s => s.Length > 3);

// Hakusyntaksilla
IEnumerable<string> result2 = from string w in oldArray
                              where w.Length > 3
                              select w;

foreach (string s in result1.Union(result2))
    Console.Write(s);
```

Esimerkissä 46 haetaan ei-geneerisestä ArrayList-tyyppisestä kokoelmasta kaikki merkkijonot, joiden pituus on suurempi kuin kolme, käyttäen hakusyntaksia ja metodisyntaksia. Ennen kuin esimerkin haussa voidaan käyttää Where-operaattoria kokoelmalle, sen sisältämät elementit täytyy tyyppimuuntaa String-tyyppisiksi Cast-operaattorilla.

OfType

OfType-operaattori vastaa Cast-standardihakuoperaattoria, eli se myös suorittaa IEnumerable-tyyppisen sekvenssin elementeille tyyppimuunnoksen ja palauttaa vahvasti tyypitetyn IEnumerable<T>-tyyppisen sekvenssin. OfType-operaattori eroaa Cast-operaattorista siinä, että se palauttaa lähdesekvenssistä vain ne elementit, joille tyyppimuunnos on mahdollinen.

Esimerkki 47: OfType-standardihakuoperaattorin käyttö.

```
ArrayList oldArray = new ArrayList();
oldArray.Add("Hello ");
oldArray.Add("C# ");
oldArray.Add(null);
oldArray.Add(3.0);
oldArray.Add("world!");

IEnumerable<string> result = oldArray
    .OfType<string>()
    .Where( w => w.Length > 2 );

foreach (string s in result)
    Console.Write(s);
```

Esimerkissä 47 luodaan ArrayList-tyyppinen kokoelma, johon sijoitetaan merkkijonoja ja desimaaliluku. Kokoelmasta haetaan pelkästään String-tyyppiset merkkijonot joiden pituus on yli kaksi merkkiä.

Jos esimerkissä olisi käytetty Cast-operaattoria OfType-operaattorin sijaan, olisi aiheutunut InvalidCastException-poikkeus, koska oldArray-kokoelma sisältää sekä String- että Double-tyyppisiä olioita. Esimerkistä nähdään myös, että OfType-operaattori ohittaa kokoelmaan sijoitetun null arvon.

5.10 Yhdistämisoperaatiot

Yhdistämisoperaatiolla voidaan liitää kaksi sekvenssiä toisiinsa.

Concat

Concat-standardihakuoperaattori yhdistää kaksi sekvenssiä yhdeksi sekvenssiksi, joka sisältää molempien sekvenssien kaikki elementit.

Esimerkissä 48 luodaan kokonaislukusekvenssi, joka liitetään itsensä jatkoksi Concat-operaattorilla.

Esimerkki 48: Concat-standardihakuoperaattorin käyttö.

```
int[] array = { 1, 2, 3 };

IEnumerable<int> result = array.Concat(array);

foreach (int i in result)
    Console.Write("{0} ", i);
```

6 Ei-viivästetyt standardihakuoperaattorit

Tässä luvussa esitellään ei-viivästetyt standardihakuoperaattorit jaoteltuina käyttötarkoituksen mukaan. Esimerkeissä käytetään samaa tietolähdettä kuin viivästettyjen standardihakuoperaattoreiden esittelyssä, joka on esitelty liitteessä 1.

6.1 Määritysoperaatiot

Määritysoperaatiot palauttavat totuusarvon, joka kertoo, täyttävätkö lähdesekvenssin kaikki elementit tai osa elementeistä annetun ehdon.

All

All-operaattori palauttaa totuusarvon siitä, täyttävätkö kaikki lähdesekvenssin elementit Func<TSource, bool>-tyyppisenä predicate-delegaattina annetun ehdon.

Esimerkki 49: All-standardihakuoperaattorin käyttö.

```
int[] array = { 5, 6, 7, 5, 9 };

bool result = array.All(n => n > 4);

Console.WriteLine(result);
```

Esimerkissä 49 luodaan kokonaislukutaulukko ja tarkistetaan All-operaattorilla, ovatko kaikki taulukon luvut suurempia kuin neljä.

Any

Any-operaattori toimii kahdella hieman erilaisella tavalla. Any-operaattori joko palauttaa totuusarvon siitä, onko lähdesekvenssissä yhtään elementtiä, tai totuusarvon siitä, täyttääkö mikään lähdesekvenssin elementeistä Func<TSource, bool>-tyyppisenä delegaattina annetun ehdon.

Esimerkki 50: Any-standardihakuoperaattorin käyttö.

```
int[] array = { 1, -2, 3};
if (array.Any())
{
    Console.WriteLine("array contains elements.");
}

bool result = array.Any(n => n < 0);
Console.WriteLine("array contains numbers below zero: {0}", result);
```

Esimerkissä 50 esitellään molempien Any-operaattorin versioiden käyttö. Esimerkissä ensiksi testataan Any-operaattorin ensimmäisellä versiolla, sisältääkö kokonaislukutaulukko array yhtään elementtiä, jos totta, niin tulostetaan tieto siitä. Seuraavaksi testataan Any-

operaattorin toisella versiolla, sisältääkö kokonaisluku-taulukko yhtään alle nollan olevaa lukua.

Contains

Contains-operaattori palauttaa totuusarvon siitä, sisältääkö lähdesekvenssi parametrina annetun elementin. Contains-operaattorille voidaan antaa lisäparametrina IEqualityComparer<TSource>-tyyppinen vertailuolio, jota käytetään yhdenvertaisuuden tarkistamiseen oletusvertailuoperaattorin sijaan.

Jos Contains-operaattorille annettu lähdesekvenssi toteuttaa geneerisen kokoelman ICollection<T>-rajapinnan, niin Contains käyttää sisäisesti kokoelman Contains-metodia.

Esimerkki 51: Contains-standardihakuoperaattorin käyttö.

```
int[] array = {0, 1, 2, 3};

bool result = array.Contains(3);

Console.WriteLine("array contains value 3: {0}", result);
```

Esimerkissä 51 tarkistetaan Contains-operaattorilla, sisältääkö array-kokonaislukutaulukko numeron kolme, ja tulostetaan tieto siitä.

6.2 Yhdenvertaisuusoperaatiot

Yhdenvertaisuusoperaatioilla voidaan tarkistaa, onko kaksi sekvenssiä yhdenvertaisia. Kahdesta sekvenssistä voidaan todeta, että ne ovat yhdenvertaiset, jos niiden vastaavat elementit ovat yhdenvertaisia ja sekvensseissä on sama määrä elementtejä.

SequenceEqual

SequenceEqual-standardihakuoperaattori määrittää, ovatko kaksi sekvenssiä yhdenvertaiset. SequenceEqual käyttää elementtien yhdenvertaisuuden tarkistukseen oletuksena elementtien oletusvertailuoperaattoria tai erikseen annettavaa IEqualityComparer<TSource>-rajapinnan toteuttavaa vertailuolioita.

SequenceEqual-operaattori palauttaa totuusarvon siitä, ovatko lähdesekvenssi ja parametrina annettu sekvenssi yhdenvertaiset.

Esimerkki 52: SequenceEqual-standardihakuoperaattorin käyttö.

```
IEnumerable<int> seq1 = Enumerable.Range(1, 10);
IEnumerable<int> seq2 = Enumerable.Range(1, 10);

Console.WriteLine("Sequences are equal: {0}",
    seq1.SequenceEqual(seq2));
```

Esimerkissä 52 luodaan kaksi kokonaislukusekvenssiä Range-operaattorilla ja tarkistetaan, ovatko ne yhdenvertaisia.

6.3 Elementtioperaatiot

Elementtioperaatiot palauttavat yhden tietyn elementin lähdesekvenssistä, jos lähdesekvenssistä ei löydy kyseistä elementtiä niin operaattorit, joko heittävät System.InvalidOperationException-poikkeuksen tai palauttavat oletusarvon.

Oletusarvon palauttavat standardihakuoperaattorit tunnistaa siitä, että niiden nimen lopussa on ”OrDefault”. Oletusarvo on referenssi- ja Nullable-tyypeille aina null ja arvotyypeille tyypin oletusarvo.

ElementAt ja ElementAtOrDefault

ElementAt- ja ElementAtOrDefault-operaattorit palauttavat lähdesekvenssistä annetun indeksiluvun mukaisen elementin. Indeksiluku on nollapohjainen.

ElementAt aiheuttaa ArgumentOutOfRangeException-poikkeuksen, jos annettu indeksiluku on alle nollan tai suurempi tai yhtä suuri kuin sekvenssin elementtien määrä. ElementAtOrDefault palauttaa poikkeustilanteessa oletuselementin.

Jos lähdesekvenssi toteuttaa IList<T>-rajapinnan, käytetään sisäisesti listan rajapintaa indeksoidun elementin hakuun. Muissa tapauksissa sekvenssi listataan läpi kunnes, indeksoitu elementti löydetään.

Esimerkki 53: ElementAt- ja ElementAtOrDefault-standardihakuoperaattorien käyttö.

```
string word1 = ExampleData.Words.ElementAt(5);
string word2 = ExampleData.Words.ElementAtOrDefault(99) ?? "NULL";

Console.WriteLine("{0}, {1}", word1, word2);
```

Esimerkissä 53 haetaan ensiksi Words-listasta ElementAt-operaattorilla indeksiluvulla 5 oleva elementti. ElementAtOrDefault-operaattorilla yritetään hakea Words-listan 99 indeksiluvulla olevaa elementtiä ja jos sitä ei löydy, niin word2-muuttujaan sijoitetaan ”NULL” merkkijono.

First ja FirstOrDefault

First- ja FirstOrDefault-operaattorit palauttavat sekvenssin ensimmäisen elementin tai oletusarvon. Operaattoreille voidaan myös antaa ehtolause Func<TSource, bool>-tyyppisenä delegaattina, jolloin ne palauttavat ensimmäisen elementin, joka täyttää annetun ehdon.

First-operaattori aiheuttaa InvalidOperationException-poikkeuksen, jos elementtiä ei löydy ja FirstOrDefault-operaattori palauttaa elementin oletusarvon.

Esimerkki 54: First- ja FirstOrDefault-standardihakuoperaattorien käyttö.

```
string word1 = ExampleData.Words.First();
string word2 = ExampleData.Words
    .FirstOrDefault(w => w.StartsWith("i"));

Console.WriteLine("Firs word: {0}", word1);

if(word2 != null)
    Console.WriteLine("First word starting with 'i': {0}", word2);
```

Esimerkissä 54 haetaan Words-listan ensimmäinen sana First-operaattorilla ja FirstOrDefault-operaattorilla haetaan Words-listasta ensimmäinen sana, joka alkaa i-kirjaimella.

Last ja LastOrDefault

Last- ja LastOrDefault-operaattorit vastaavat toiminnaltaan First- ja FirstOrDefault-operaattoreita, mutta ne palauttavat sekvenssin viimeisen elementin tai viimeisen elementin, joka täyttää Func<TSource, bool>-tyyppisenä delegaattina annetun ehdon.

Jos lähdesekvenssi on tyhjä tai mikään elementti ei täytä annettua ehtolauseketta, Last-operaattori heittää InvalidOperationException-poikkeuksen ja LastOrDefault-operaattori palauttaa elementin oletusarvon.

Esimerkki 55: Last- ja LastOrDefault-standardihakuoperaattoreiden käyttö.

```
string word1 = ExampleData.Words.Last();
string word2 = ExampleData.Words
    .LastOrDefault(w => w.StartsWith("i"));

Console.WriteLine("Last word: {0}", word1);

if (word2 != null)
    Console.WriteLine("Last word starting with 'i': {0}", word2);
```

Esimerkissä 55 haetaan Words-listan viimeinen sana Last-operaattorilla ja LastOrDefault-operaattorilla listan viimeinen sana, joka alkaa i-kirjaimella.

Single ja SingleOrDefault

Single- ja SingleOrDefault- operaattorit palauttavat lähdesekvenssin ainoan elementin tai elementin, joka täyttää Func<TSource, bool>-tyyppisenä delegaattina annetun ehdon. Single- ja SingleOrDefault-operaattoreita käytetään silloin kun halutaan hakea vain yksi elementti tietolähteestä.

Jos lähdesekvenssi sisältää enemmän kuin yhden elementin tai useampi kuin yksi elementti täyttää annetun ehdon, Single- ja SingleOrDefault-operaattorit heittävät InvalidOperationException-poikkeuksen.

Single eroaa SingleOrDefault-operaattorista sillä, että se heittää InvalidOperationException-poikkeuksen myös silloin kun lähdesekvenssi on tyhjä tai mikään elementti ei täytä annettua ehtoa, vastaavissa poikkeustilanteissa SingleOrDefault palauttaa oletusarvon.

Esimerkki 56: Single- ja SingleOrDefault-standardihakuoperaattoreiden käyttö.

```
Product product = ExampleData.Products.Single(p => p.Id == 1);

Category category = ExampleData.Categories
    .SingleOrDefault( c => c.Id == 6);

Console.WriteLine("Prduct id: {0} - {1}", product.Id,
    product.Name);

if (category == null)
    Console.WriteLine("Category not found.");
else
    Console.WriteLine("Category id: {0} - 1", category.Id,
        category.Name);
```

Esimerkissä 56 haetaan Single-operaattorilla Products-listasta tuote, jonka id on 1 ja tulostetaan tuotteen id ja nimi. Seuraavaksi yritetään hakea SingleOrDefault-operaattorilla Categories-listasta kategoria id luvulla 6. Jos kategoriaa ei löydy, tulostetaan teksti ”Category not found”, muulloin tulostetaan kategorian id ja nimi.

6.4 Muunnosoperaatiot

ToArray ja ToList

ToArray-standardioperaattori palauttaa IEnumerable<T>-tyyppisestä lähdesekvenssistä T-tyyppisen taulukon ja ToList-standardihakuoperaattori palauttaa lähdesekvenssistä List<T>-tyyppisen kokoelman.

Esimerkissä 57 haetaan Products-listasta tuotteiden hinnat desimaalitaulukkoon ToArray-operaattorilla ja tulostetaan taulukon kolmas desimaaliluku.

Esimerkki 57: ToArray-standardihakuoperaattorin käyttö.

```
decimal[] prices = ExampleData.Products
    .Select(p => p.Price)
    .ToArray();

Console.Write(prices[2]);
```

Esimerkki voitaisiin toteuttaa täysin samanlaisena ToList-operaattorilla, jolloin prices-muuttujan tyyppi pitäisi vain vaihtaa List<decimal>-tyyppiseksi.

ToDictionary

ToDictionary-standardihakuoperaattori palauttaa Dictionary<TKey, TResult>-tyyppisen kokoelman IEnumerable<TResult>-tyyppisestä lähdesekvenssistä.

ToDictionary-operaattori käyttää Dictionary-kokoelman avainten vertailuun, joko oletusvertailuoperaattoria tai erikseen annettua IEqualityComparer<T>-rajapinnan toteuttavaa vertailuoliota.

ToDictionary-operaattorille annetaan parametrina Func<TSource, TKey>-tyyppinen keySelector-delegaatti, jonka tarkoitus on palauttaa lähdesekvenssin elementeistä Dictionary-kokoelman avaimet, joiden vastaavina arvoina lähdesekvenssin elementit toimivat. Operaattorille voidaan antaa lisäparametrina Func<TSource, TElement>-tyyppinen elementSelector-delegaatti, jolla voidaan tehdä projektio kokoelmaan valittaville elementeille.

Esimerkki 58: ToDictionary-standardihakuoperaattorin käyttö.

```
Dictionary<string, decimal> result = ExampleData.Products
    .ToDictionary( p => p.Name, p => p.Price);

foreach (KeyValuePair<string, decimal> kp in result)
    Console.WriteLine("{0} - {1}", kp.Key, kp.Value);
```

Esimerkissä 58 muodostetaan Products-listasta Dictionary<string, decimal>-tyyppinen kokoelma ja tulostetaan sen avain-arvoparit. Kokoelman avain-arvopareina toimivat tuotteen nimi ja hinta.

ToLookup

ToLookup-standardihakuoperaattori palauttaa Lookup<TKey, TResult>-tyyppisen kokoelman IEnumerable<TResult>-tyyppisestä lähdesekvenssistä.

Lookup-kokoelma vastaa Dictionary-kokoelmaa, erona se, että Dictionary-kokoelman avain-arvoparien sijaan Lookup-kokoelmassa yhtä avainta vastaa kokoelma arvoja. Lookup-kokoelman elementit esitetään IGrouping<TKey, TElement>-tyyppisinä olioina.

ToLookup-operaattori ryhmittelee lähdesekvenssin elementit avainten perusteella kuten GroupBy-standardihakuoperaattori, ja se käyttää avainten vertailussa oletusvertailuoperaattoreita tai erikseen parametrina annettavaa IEqualityComparer<T>-rajapinnan toteuttavaa vertailuoliota.

ToLookup-standardihakuoperaattorille annetaan parametrina Func<TSource, Tkey>-tyyppinen keySelector-delegaatin, jonka tarkoitus on palauttaa lähdesekvenssin

elementeistä Lookup-kokoelman avaimet ja sille voidaan myös antaa lisäparametrina Func<TSource, TElement>-tyyppinen elementSelector-delegaatti, jolla tehdään projektio kokoelmaan valituille elementeille.

Esimerkki 59: ToLookup-standardihakuoperaattorin käyttö.

```
ILookup<char, string> result = ExampleData.Products
    .ToLookup(p => p.Name[0], p => p.Name);

foreach (IGrouping<char, string> productGroup in result)
{
    Console.WriteLine(productGroup.Key);
    foreach (string productName in productGroup)
        Console.WriteLine("   {0}", productName);
}
```

Esimerkissä 59 ryhmitellään Products-listan tuotteet nimen ensimmäisen kirjaimen mukaan ja tallennetaan ryhmät Lookup<char, string>-tyyppiseen kokoelmaan. Tuotteen nimen ensimmäinen kirjain toimii ryhmän avaimena. Esimerkissä tulostetaan ryhmän avain ja ryhmän sisältämät tuotteiden nimet.

6.5 Koosteoperaatiot

Koosteoperaatiolla voidaan laskea tai muodostaa yksittäinen arvo arvojoukosta.

Aggregate

Aggregate-operaattorilla voidaan suorittaa akkumulaatiofunktio sekvenssin elementeille. Sille annetaan parametrina Func<TAccumulate, Tsource, TAccumulate>-tyyppinen delegaatti, joka suoritetaan kumulatiivisesti jokaiselle sekvenssin elementille, siten että seuraava delegaatin kutsukerta saa parametrina aina edellisen kutsukerran palautusarvon ja käsiteltävän elementin. Operaattorille voidaan myös antaa alkuarvo ja Func<TAccumulate, TResult>-tyyppinen delegaatti projektion suorittamiseksi lopputuloksesta.

Esimerkki 60: Aggregate-standardihakuoperaattorin käyttö.

```
string result = ExampleData.Words
    .Take(10)
    .Aggregate((acc, next) => acc + " " + next);

Console.WriteLine(result);
```

Esimerkissä 60 haetaan Words-taulukosta kymmenen ensimmäistä merkkijonoa, joista muodostetaan yksi merkkijono Aggregate-standardihakuoperaattorilla.

Average ja Sum

Average-standardihakuoperaattori palauttaa lähdesekvenssin numeeristen arvojen keskiarvon ja Sum-standardihakuoperaattori palauttaa lähdesekvenssin numeeristen arvojen summan.

Operaattoreille voidaan antaa lisäparametrina Funct<TSource, TNumeric>-tyyppinen selector-delegaatti, jonka tarkoitus on projisoida lähdesekvenssin elementeistä laskentaan käytettävä numeerinen arvo, jos lähdesekvenssi ei itsessään sisällä numeerisia arvoja.

Operaattorien palautusarvon tyyppi riippuu numeerisen lähdesekvenssin elementtien tyypistä tai selector-delegaatin palauttamasta TNumeric-tyypistä. Average operaattori tyyppimuuntaa keskiarvon laskentaa varten kokonaislukutyypit double- tai double?-tyypeiksi.

Esimerkki 61: Average- ja Sum-standardihakuoperaattorien käyttö.

```
double avg = ExampleData.Words.Average(w => w.Length);
int sum = ExampleData.Words.Sum(w => w.Length);

Console.WriteLine("Average word length: {0}", avg);
Console.WriteLine("Sum of characters: {0}", sum);
```

Esimerkissä 61 lasketaan Words-taulukon merkkijonojen pituuksien keskiarvo Average-operaattorilla ja merkkijonojen pituuksien summa Sum-operaattorilla.

Max ja Min

Max-standardihakuoperaattori palauttaa lähdesekvenssin suurimman elementin ja Min-standardihakuoperaattori palauttaa lähdesekvenssin pienimmän elementin. Operaattorit käyttävät elementtien vertailussa annetun tyypin toteuttamaa IComparable<T>- tai IComparable-rajapintaa.

Operaattoreille voidaan antaa lisäparametrina Func<TSource, TResult>-tyyppinen selector-delegaatti, jonka tarkoitus on palauttaa lähdesekvenssin elementistä vertailuun käytettävä arvo.

Esimerkki 62: Max- ja Min-standardihakuoperaattorien käyttö.

```
decimal maxPrice = ExampleData.Products.Max(prod => prod.Price);
decimal minPrice = ExampleData.Products.Min(prod => prod.Price);

Console.WriteLine("Highest price: {0}", maxPrice);
Console.WriteLine("Lowest price: {0}", minPrice);
```

Esimerkissä 62 haetaan Products-listasta kalleimman tuotteen hinta Max-standardihakuoperaattorilla ja halvimman tuotteen hinta Min-operaattorilla.

Count ja LongCount

Count- ja LongCount-standardihakuoperaattorit palauttavat sekvenssin elementtien lukumäärän tai niiden sekvenssin elementtien lukumäärän, jotka täyttävät Func<TSource, bool>-tyyppisenä delegaattina annetun ehdon.

Count- ja LongCount-standardihakuoperaattorit toimivat samalla tavalla. Niiden erona on vain se, että Count-operaattorin palautusarvo on tyyppiä int ja LongCount-operaattorin palautusarvo on tyyppiä long.

Jos lähdesekvenssi on tyhjä, palauttavat Count- ja LongCount-operaattorit aina arvon nolla.

Esimerkki 63: Count-standardihakuoperaattorin käyttö.

```
int result = ExampleData.Words.Count(w => w.Length < 3);
Console.WriteLine(result);
```

Esimerkissä 63 lasketaan niiden Words-taulukon merkkijonojen lukumäärä, jotka ovat alle kolme merkkiä pitkiä.

7 Linq To SQL

Tässä luvussa esitellään Linq To SQL -komponentti ja sen käyttämän tietomallin rakentaminen ja hakujen käyttö tietokantakyselyissä.

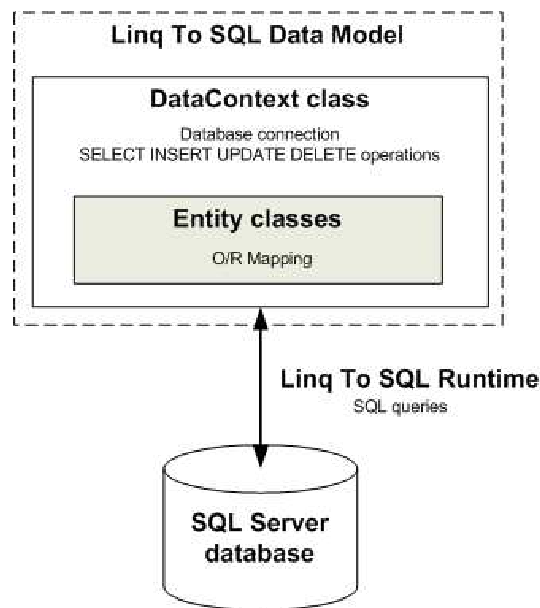
Linq to SQL on Microsoftin kehittämä olio-relaatiomallinnus SQL Server -relaatiotietokannalle, joka hyödyntää hakuja tietokantakyselyissä.

Olio-relaatiomallinnus, lyhyesti ORM (Object-Relational Mapping), on ohjelmointitekniikka, jossa muutetaan tietoa relaatiotietokantojen ja olio-ohjelmointikielien yhtyeensopimattomien tietotyyppijärjestelmien välillä. [16, s. 4.]

Linq To SQL toteuttaa korkean tason olio-relaatiomallinnuksen, joten kaikkiin mallinnuksen yksityiskohtiin ei voida vaikuttaa. Vaihtoehtona komponentille on Entity Framework, joka tarjoaa monipuolisemman ja osaltaan monimutkaisemman lähestymistavan vastaavaan olio-relaatiomallinnukseen.

Linq to SQL etu jo olemassa oleviin ORM-komponentteihin, kuten NHibernaten, on standardihakuoperaattoreiden käyttäminen tiedonhakuun tietomallista. Useimmat ORM-komponentit eivät pysty tarjoamaan yhtä monipuolisia hakumahdollisuuksia mallinnettuun tietomalliin, ja niiden omat hakurajapinnat voivat olla vaikeita omaksua. [10.]

Kuvassa 4 on esitetty Linq TO SQL -komponentin rakenne, josta voidaan erottaa komponentin kolme keskeisintä osaa: tietomalli, Linq To SQL -suoritusvaihe ja tietokanta.



Kuva 4: Linq TO SQL komponentin rakenne

Tietomalli (kuvassa Data Model) koostuu DataContext-luokasta ja entiteettiluokista. DataContext-luokka vastaa tietokantayhteyksistä, tietomallin muutosten seurannasta, muutosten päivittämisestä tietokantaan ja monesta muusta itse tietokannan käyttöön liittyvästä toiminnasta.

Entiteettiluokat (kuvassa Entity classes) vastaavat olio-relaatiomallinnuksesta, jossa yleensä yksi entiteettiluokka mallintaa yhden tietokantataulun ja entiteettiluokan ominaisuudet vastaavat kyseisen taulun kenttiä.

Komponentin käyttämä tietomalli generoidaan yleensä jollain työkalulla suoraan valmiista tietokannasta tai se voidaan tehdä käsin. SQL Server -tietokanta voidaan myös luoda tietomallin pohjalta.

Tietomalli toteuttaa IQueryable<T>-rajapinnan, joka mahdollistaa standardihaku-operaattoreiden käytön tietokantakyselyissä.

Linq To SQL -suoritusvaihe vastaa hakujen muuntamisesta SQL-kyselyiksi, jotka annetaan tietokantapalvelimelle suoritettavaksi ja kyselyiden tuloksien muuntamisen takaisin olioiksi.

Esimerkki 64: Linq To SQL -haun suoritus, jossa kirjataan lokiin Linq To SQL -suoritusvaiheen muodostama SQL-lause.

```
using (DemoDataContext dx = new DemoDataContext())
{
    dx.Log = Console.Out;

    foreach (Category c in dx.Categories)
        Console.WriteLine("{0} - {1}", c.Id, c.Name);
}
```

Esimerkissä 64 esitellään haun suoritus ja lokin kirjoitus. Esimerkissä haetaan esimerkkitietokannan Category-taulun sisältö ja tulostetaan Linq To SQL -suoritusvaiheen hausta muodostama SQL-lause. Esimerkki tulostaa seuraavaa:

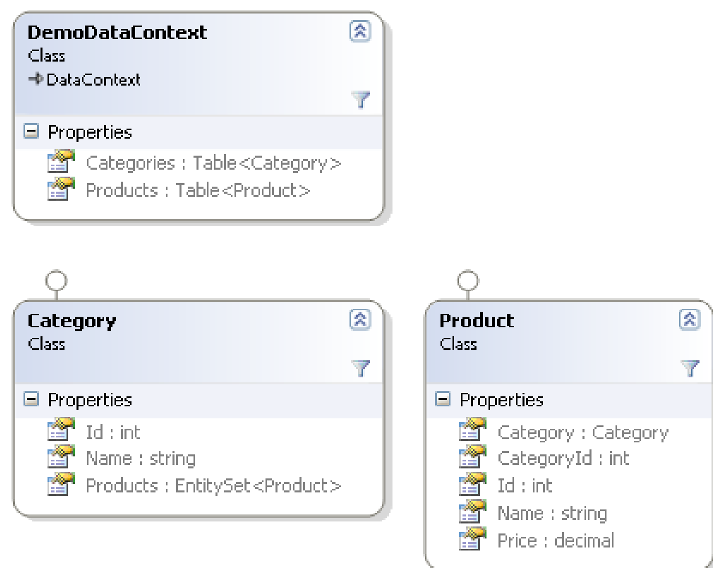
```
SELECT [t0].[Id], [t0].[Name]
FROM [Category] AS [t0]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build:
3.5.30729.1

2 - Cars
3 – Books
```

Esimerkin tulostuksen alussa nähdään Linq To SQL -suoritusvaiheen muodostama SQL-kysely, joka annettiin tietokannan suoritettavaksi.

7.1 Esimerkkitietomalli ja vastaava esimerkkitietokanta

Linq To SQL -komponenttia käsittelevissä esimerkeissä käytetään Visual Studio 2008 Object Relational Designer -työkalulla generoitua tietomallia. Kuvassa 5 on esitelty esimerkkitietomallin luokkakaavio, joka on luotu Visual Studio 2008 -luokkakaaviotyökalulla.



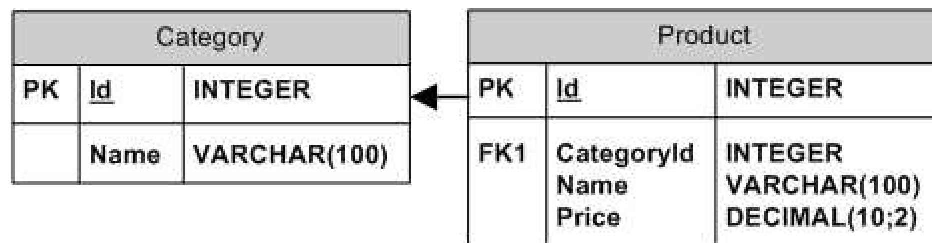
Kuva 5: Linq To SQL -esimerkkitietomalli

Esimerkkitietomalli sisältää DemoDataContex-luokan, joka periytyy DataContext-luokasta ja kaksi entiteettiluokkaa, Category ja Product, jotka esittävät tuotekategoriaa ja tuotetta.

Kategorialla on ominaisuuksina tunnusluku, nimi ja kokoelma kategoriaan kuuluvista tuotteista. Kategorian tunnusluvulla voidaan identifioida yksittäinen kategoria.

Tuotteella on ominaisuuksina tunnusluku, nimi, hinta ja kategorian tunnusluku, johon tuote kuuluu. Tuotteen tunnusluvulla voidaan identifioida yksittäinen tuote ja tuotteen kategorian tunnusluvulla voidaan määrittää mihin kategoriaan tuote kuuluu. Tuotteella on myös viittaus siihen kategoriaolioon, johon se kuuluu.

Esimerkkitietomallia vastaa kuvassa 6 esitelty esimerkkitietokanta, joka koostuu kahdesta taulusta Category ja Product.



Kuva 6: Esimerkkitietokannan taulut

Esimerkkitietokannan Product-taulusta on 1-N-relaatio Category-tauluun.

Esimerkkitietomallin käyttämä tietokanta on luotu käyttämällä DataContext-luokan CreateDatabase-metodia, jolla voidaan luoda tietokanta tietomallin pohjalta. Esimerkissä 65 on esitelty esimerkkitietokannan luominen tietomallin pohjalta. Tietokannan luonnin yhteydessä tietokantaan lisätään kaksi kategoriaa, ”Books” ja ”Cars”, ja niille molemmille tuotteita.

Esimerkki 65: Esimerkkitietokannan luonti tietomallin pohjalta.

```
using (DemoDataContext dx = new DemoDataContext())
{
    if (!dx.DatabaseExists())
    {
        // Create database
        dx.CreateDatabase();

        // Insert example data
        Category cars = new Category
        {
            Name = "Cars",
            Products = {
            new Product { Name = "BMW", Price = 100000 },
            new Product { Name = "Ferrari", Price = 200000 },
            new Product { Name = "McLaren", Price = 150000 },
            new Product { Name = "Lada", Price = 6500 }
        }
        };
        dx.Categories.InsertOnSubmit(cars);

        Category books = new Category
        {
            Name = "Books",
            Products = {
            new Product { Name = "Pro Linq", Price = 23.99M },
            new Product { Name = "Xml", Price = 15 },
            new Product { Name = "Linq To Objects", Price = 19.99M },
            new Product { Name = "Professional C#", Price = 9.99M }
        }
        };
        dx.Categories.InsertOnSubmit(books);

        dx.SubmitChanges();
    }
}
```

7.2 Olio-relaatiomallinnus

Tässä luvussa esitellään Linq To SQL -komponentin olio-relaatiomallinnus yleisellä tasolla. DataContext-luokan tarkempi toiminnallisuus esitellään omassa kappaleessa. Olio-relaatiomallinnuksen esittelyssä käytetään esimerkkitietokantaa ja siitä tehtyä esimerkkitietomallia, jotka on esitelty luvussa 7.1.

Taulukossa 1 on esitetty Linq To SQL olio-relaatiomallinnuksen tietomallin elementtien suhteet vastaaviin relaatiomallin elementteihin.

Taulukko 1: Linq To SQL -tietomallin ja relaatiotietokannan välisen olio-relaatiomallinnuksen keskeiset elementit ja niiden suhteet toisiinsa. [11.]

LINQ to SQL Object Model	Relational Data Model
DataContext class	Database
Entity class	Table
Class member	Column
Association	Foreign-key relationship
Method	Stored procedure or Function

Olio-relaatiomallinnuksessa DataContext-luokasta periytyvä luokka vastaa kokonaista relaatiotietokantaa tai osaa siitä. Se sisältää tietokannan taulut geneerisinä Table<T>-

tyyppisinä ominaisuuksina, jossa tyyppi T vastaa entiteettiluokkaa. Table<T>-luokka toteuttaa IQueryable<T>-rajapinnan.

Esimerkiksi DemoDataContext-luokka sisältää tietokannan Category- ja Product-taulut, jotka ovat määritelty sen seuraavina ominaisuuksina:

```
public Table<Product> Products
public Table<Category> Categories
```

Jos esimerkiksi halutaan hakea tietokannasta tuotteita, käytetään haun kohteena DemoDataContext-olion Products-kokoelmaa.

Tietokannan sisältämät proseduurit ja funktiot mallinnetaan DataContext-luokan metodeiksi, jolloin niiden käyttö vastaa luokan metodikutsuja.

Tietokannan taulut mallinnetaan omina luokkina, joista käytetään entiteettiluokka nimitystä. Yksi entiteettiluokka vastaa yhtä tietokannan tauluista tai osaa yhdestä tietokannan taulusta. Linq To SQL ei tue entiteettiluokkia, jotka koostuvat useasta eri taulusta. Luokan ominaisuudet vastaavat tietokantataulun kenttiä. Linq To SQL tukee entiteettiluokkien perintää, mutta rajoitteena on se, että kaikkien perittyjen entiteettiluokkien ominaisuuksien pitää olla samassa tietokantataulussa.

Entiteettiluokan ominaisuuksien ja luokkien välisten assosiaatioiden olio-relaatio-mallinnukseen tarvittavat tiedot määritellään entiteettiluokkien attribuuteilla tai erillisessä XML muotoisessa mallinnustiedostossa.

Esimerkki 66: Esimerkki Linq To SQL olio-relaatiomallinnuksen käyttämistä attribuuteista.

```
[Column(Storage="_Id", AutoSync=AutoSync.OnInsert,
    DbType="int IDENTITY", IsPrimaryKey=true, IsDbGenerated=true,
    UpdateCheck=UpdateCheck.Never)]
public int Id
{
    get
    {
        return this._Id;
    }
}
```

Esimerkissä 66 esitellään Product-entiteettiluokan Id-ominaisuus ja sen attribuutit, jotka määrittävät Id-ominaisuuden suhteen esimerkkitietokannan Product-taulun Id-kenttään. Esimerkiksi Column-attribuutin DbType-parametri määrittää tietokannan käyttämän tyypin, joka on esimerkissä int-tyyppinen identiteettikenttä ja IsPrimaryKey-parametri määrittää, onko ominaisuus tietokantataulun pääavain.

Linq To SQL -olio-relaatiomallinnuksessa tietokantataulujen väliset relaatiot mallinnetaan entiteettiluokkien välisinä assosiaatioina, joissa toisiinsa liittyvillä luokilla on ominaisuutena referenssi toiseen entiteettiluokkaan tai kokoelma entiteettiluokkia.

Relaatiotietokannan 1-1-relaatiosuhde mallinnetaan entiteetttiluokan EntityRef<T>-tyyppisenä ominaisuutena, jossa T on referoidun entiteettiluokan tyyppi ja 1-N-relaatio mallinnetaan entiteettiluokan EntitySet<T>-tyyppisenä ominaisuutena, joka on kokoelma T tyyppisiä entiteettiolioita. Esimerkiksi Category-luokalla on seuraava Products-ominaisuus:

```
private EntitySet<Product> _Products;

public EntitySet<Product> Products
{
    get { return this._Products; }
    set { ... }
}
```

Ominaisuus sisältää kaikki Category-entiteettiolioon assosioidut Products-entiteettioliot, eli ne vastaavat Products-tietokantataulun rivit, joista on viite kyseiseen Category-taulun riviin.

Vastaavasti Product-luokalla on referenssi Category-luokkaan johon se kuuluu. Product-luokan 1-1 suhteessa oleva referenssi Category-luokkaan on mallinnettu Product-luokan ominaisuudella Category:

```
private EntityRef<Category> _Category;

public Category Category
{
    get { return this._Category.Entity; }
    set { ... }
}
```

Entiteettiluokkien väliset assosiaatiot helpottavat huomattavasti ohjelmointityötä. Assosiaatiot mahdollistavat helpon relaatioiden välisen liikkumisen äitioliosta lapsiolioon ja toisinpäin, ilman että tarvitaan erillisiä liitosoperaatioita.

7.3 Tietomallin lähdekoodin generointi

Tietomalli voidaan generoida kahdella eri työkalulla, komentorivipohjaiselle SLQMetal-työkalulla tai sitten graafisella käyttöliittymäpohjaisella Visual Studio 2008 Object Relational Designer -työkalulla.

Molemmat työkalut tuottavat lopputuloksena ohjelmakoodilistauksen, joka sisältää käytettävän DataContext-luokan ja entiteettiluokkien lähdekoodin.

Tietomallin voi myös kirjoittaa käsin, mikä ei ole kuitenkaan suuren työmäärän ja mahdollisten virheiden vuoksi kannattavaa.

SQLMetal-työkalu

SQLMetal-ohjelma on komentorivipohjainen työkalu, jolla voidaan generoida tietomalli valmiista tietokannasta. Ohjelma kuuluu Windows SDK -kehitystyökaluihin, jotka asennetaan Visual Studio 2008:n asennuksen yhteydessä, ja se löytyy oletuksena seuraavasta kansiosta [12.]:

```
drive:\Program Files\Microsoft SDKs\Windows\vn.nn\bin
```

SQLMetal-ohjelmalla voidaan generoida kolmenlaisia tiedostoja:

- Pelkkä tietomallin lähdekooditiedosto, jossa olio-relaatiomallinnukseen tarvittavat tiedot ovat määritetty entiteettiluokkien attribuuteilla.
- XML-muotoinen DBML (DataBase Markup Language) tiedosto, jota voidaan muokata käsin tai graafisella Object Relational Designer -työkalulla. DBML-tiedostosta voidaan muokkauksen jälkeen luoda lähdekooditiedosto SQLMetal- tai designer -työkaluilla.
- Lähdekoodi- ja XML-mallinnustiedostopari, jossa lähdekooditiedosto sisältää vain tietomallin entiteettiluokkien lähdekoodit ja olio-relaatiomallinnukseen tarvittavat tiedot on määritelty erillisessä XML-tiedostossa.

Lähdekooditiedoston generointi valmiista tietokannasta on helpoin ja nopein tapa saada tietokanta mallinnettua, mutta tällöin ei ole kovin suurta mahdollisuutta vaikuttaa mallinnukseen. Esimerkiksi tietokanta saattaa käyttää erikoista taulujen nimeämiskäytäntöä, jolloin pitäisi pystyä vaikuttamaan generoitavien entiteettiluokkien nimiin.

DBML-tiedoston generointi on hyödyllistä silloin kun mallinnetaan isoa tietokantaa, jolloin saadaan helposti tuotettua valmis DBML-tiedosto, jota voidaan jatkokäsiteltäväksi graafisella työkalulla.

Erillinen XML-mallinnustiedostoa käytettäessä entiteettiluokkien lähdekoodin ei generoida ollenkaan tietokannasta riippuvaa ohjelmakoodia, jolloin on esimerkiksi mahdollista muuttaa tietokantatoteutusta ilman, että lähdekoodia joudutaan kirjoittamaan uusiksi.

SQLMetal-ohjelman rajoitteena on se, että se generoi aina tietomallin koko tietokannasta. Jos halutaan esimerkiksi generoida tietomalliin vain osa tietokannan tauluista, joudutaan käyttämään muita työkaluja.

SQLMetal-ohjelmalle voidaan antaa useita erilaisia parametreja, joilla voidaan vaikuttaa ohjelman generoimaan lopputulokseen. Tarkemmat tiedot ohjelman toiminnasta löytyvät ohjelman dokumentaatiosta. [12.]

Esimerkki 67: SQLMetal-ohjelman käyttö.

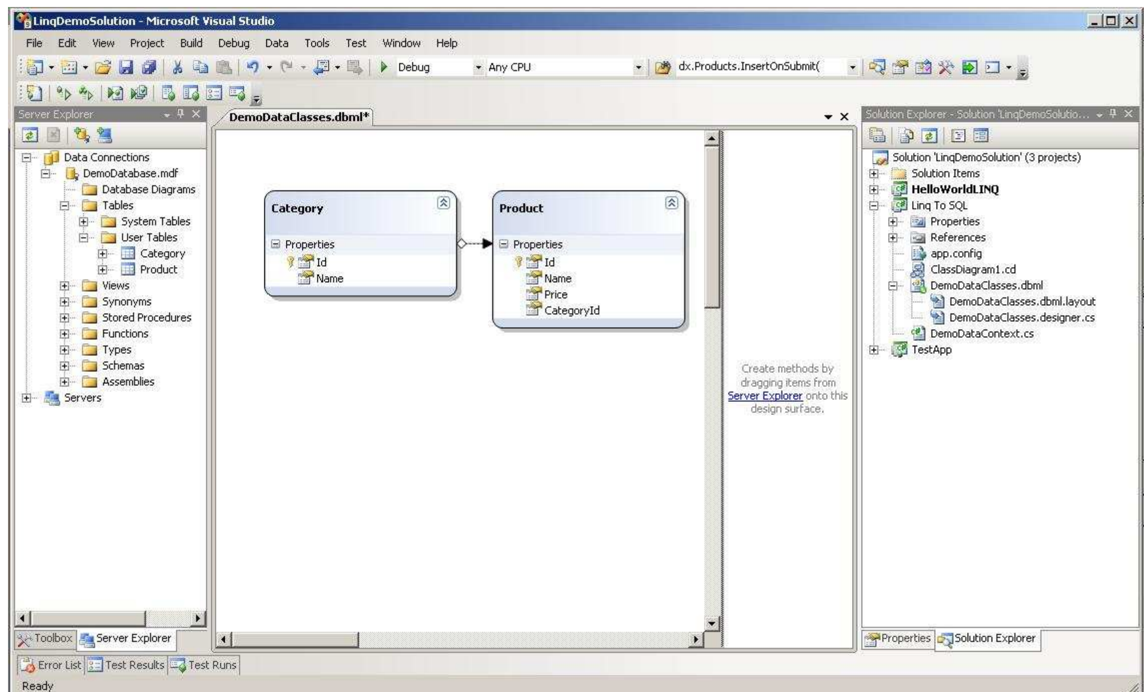
```
SqlMetal.exe DemoDatabase.mdf /code:DemoDatabase.cs  /namespace:DemoDB
/context:DemoDataContext
```

Esimerkissä 67 on esitelty SQLMetal-ohjelman käyttö. Esimerkin komennolla generoidaan ”DemoDatabase.mdf”-nimisestä SQL Server -tietokantatiedostosta tietomalli lähdekooditiedostoksi. ”/code ”-parametrilla annetaan generoitavan lähdekooditiedoston nimi. ”/namespace”-parametrilla annetaan generoitavien entiteettiluokkien ja DataContext-luokan nimiavaruus. ”/context”-parametrilla annetaan generoitavan DataContext-luokasta periytyvän luokan nimi.

Visual Studio 2008 Object Relational Designer -työkalu

Object Relational Designer on graafinen käyttöliittymäpohjainen Visual Studio 2008 -työkalu Linq To SQL -tietomallin generointiin. Työkalu mahdollistaa tietomallin rakentamisen valmiin tietokannan pohjalta tai tietomallin suunnittelun alusta lähtien ilman valmista tietokantaa. Työkalulla on mahdollista muokata suurinta osaa olio-relaatiomallintamisen määrityksiä, kuten esimerkiksi generoitavien entiteettiluokkien nimiä, entiteettiluokan ominaisuuksien nimiä, tietotyyppejä ja nimiavaruuksia.

Työkalun saa käyttöön, kun lisää projektiin uuden DBML-tiedoston, joka on Visual Studiossa nimeltä ”Linq To SQL Classes file”.



Kuva 7: Visual Studio 2008 Object Relational Designer

Kuvassa 7 on esitelty Object Relational Designer -työkalun käyttöliittymä. Työkalu vastaa toiminnaltaan muita Visual Studio 2008 -suunnittelutyökaluja, joten sen toimintaa ei esitellä kovin tarkasti. Tarkemmat ohjeet työkalun käyttöön löytyvät työkalun dokumentaatiosta. [13.]

Yleisin työkalun käyttötapa on avata Server Explorer -työkalulla (kuvassa vasemmalla) tietokantayhteys tietokantaan, joka halutaan mallintaa. Tietomallin luominen valmiista tietokannasta onnistuu viemällä Server Explorerista tietokannan tauluja designerin (kuvassa keskellä) tilaan.

Entiteettiolioiden ja DataContext-luokan ominaisuuksia voidaan muokata Properties-välilehdeltä. Uusien entiteettiolioiden luominen ja entiteettiolioiden välisten assosiaatioiden luominen onnistu käyttämällä Toolbox-välilehden työkaluja.

Solution Explorerista (kuvassa oikealla) nähdään, että DemoDataClasses.dbml-tiedostolla on alitiedosto DemoDataClasses.dbml.designer.cs, joka pitää sisällään työkalulla tehdyn tietomallin lähdekoodin. Tämä lähdekoodi käännetään normaalisti käännöksen yhteydessä. Vaikka generoitua lähdekoodia pystyy tarkistelemaan, ei siihen kannata tehdä muutoksia käsin, koska jos lähdekoodi generoidaan uudelleen, menetetään tällöin käsin tehdyt muutokset. Työkalu generoi lähdekoodin aina uudelleen, kun DBML-tiedostoon on tehty muutoksia ja tiedosto on tallennettu.

Kuvassa olevaan suunnittelutyökaluun on avattu aikaisemmin esitelty esimerkkitietomalli, joka sisältää entiteettioliot Product ja Category. Kuvasta nähdään myös entiteettien välinen assosiaatio, joka esitetään työkalussa nuolena.

Suunnittelutyökalun yksi ominaisuus on se, että se osaa luoda sovelluksen konfiguraatiotiedostoon tietokantayhteyttä varten tarvittavan ConnectionString-asetuksen ja se generoi DataContext-luokalle oletusrakentajan, joka käyttää tätä asetusta resurssitiedoston kautta. Tällöin DataContext-olion instanssia luotaessa, sille ei tarvitse erikseen välittää ConnectionString-parametria. Esimerkkitietomallissa on tehty juuri näin.

Object Relational Designerin suurin hyöty on se, että sillä voidaan suunnitella tietomalli ilman, että käytetään valmista tietokantaa pohjana. Tällöin voidaan kehittää ja suunnitella sovelluksen käyttämää tietomallia olio-ohjelmoinnin näkökulmasta, jolloin saatu lopputulos vastaa paremmin olio-ohjelmoinnin vaatimuksia. Relaatiotietokantojen suunnittelussa ei aina osata ottaa huomioon kaikkia olio-ohjelmoinnin vaatimuksia, minkä vuoksi saatetaan päätyä ratkaisuihin, jotka toimivat hyvin relaatiomallissa, mutta huonosti oliomallissa. Koska tietomallin perusteella voidaan generoida vastaava relaatiotietokanta, voidaan olla varmoja, että tietokanta vastaa suunniteltua oliomallia.

7.4 DataContext-luokka

DataContext-luokka on Linq To SQL -komponentin keskeinen tekijä. DataContext-luokan vastuulla on seuraavat toiminnallisuudet:

- tietokantayhteyden hallinta
- tietokantaoperaatioiden suoritus
- tietomallin entiteettiolioiden identiteetin seuranta
- tietomallin muutosten seuranta ja käsittely
- transaktioiden eheyden seuranta ja konfliktien hallinta.

DataContext-luokkaa voidaan käyttää sellaisenaan ohjelmoinnissa, mutta on yleisesti suositeltavaa käyttää siitä perittyä omaa luokkaa, joka on vahvasti tyypitetty vastaamaan käytettävää tietokantaa. Tietomallin lähdekoodin generoinnin yhteydessä generoidaan käytettävää tietokantaa vastaava DataContext-luokasta peritty luokka, kuten esimerkkitiemallin DemoDataContext-luokka.

DataContext-luokka vastaa tietokantayhteyden hallinnasta, joten käyttäjän ei tarvitse huolehtia yhteyden muodostamisesta tai sulkemisesta. Luokka muodostaa tietokantayhteyden ennen tietokantahaun suorittamista ja sulkee sen haun suorituksen jälkeen.

DataContex-luokka toteuttaa IDisposable-rajapinnan, joten ohjelmoijan on huolehdittava sen oikeaoppisesta poistamisesta. DataContext-luokkaa käytettäessä monisäikeisessä ympäristössä, kuten esimerkiksi Web-sovelluksissa, on myös huomioitava, että se ei ole säieturvallinen. DataContext-luokka on suunniteltu siten, että sen instantioiminen on mahdollisimman nopeaa, joten yhdellä olion instanssilla voidaan suorittaa jokin tietokantaoperaatio ja sen jälkeen poistaa instanssi muistista, ilman että sovelluksen suorituskyky kärsii.

DataContext-luokan rakentimelle annetaan parametrina tietokantayhteyden ConnectionString tai IDbConnection-rajapinnan toteuttava tietokantayhteysolio, jota se käyttää tietokantayhteyden muodostamiseen. Esimerkkitietomallin DemoDataContext-luokalle on generoitu parametriton rakennin, joka muodostaa tietokantayhteyden resurssitiedostossa olevan ConnectionStringin perusteella.

DataContext vastaa tietokantaoperaatioiden suorittamisesta, se muuntaa entiteettiolioihin kohdistetut haut SQL-tietokantahauksi ja antaa ne tietokannalla suoritettavaksi käyttäen määritettyä tietokantayhteyttä. Dynaamisesti muodostettavien SQL-lausekkeiden sijaan voidaan myös käyttää itse tehtyjä SQL-lauseita periyttämällä DataContext-luokka ja ylikirjoittamalla tarvittavat metodit.

DataContext-luokka tukee normaalien tietokantaoperaatioiden lisäksi seuraavia toimintoja:

- tietokannan luonti ja poisto CreateDatabase- ja DeleteDatabase-metodeilla
- tavallisten SQL-lauseiden suorittaminen suoraan tietokantaan ExecuteCommand-metodilla
- suoritettavien SQL-lauseiden lokiin kirjoitus Log-ominaisuuden kautta.

Tietomallin entiteettiolioiden identiteetin seuranta

DataContext-luokka vastaa entiteettiolioiden identiteetin seurannasta, millä tarkoitetaan sitä, että DataContext varmistaa tietomallia käsiteltäessä, että käsitellään aina samaa entiteettiolion instanssia. Esimerkiksi jos sama entiteettiolio haetaan tietokannasta useaan kertaan lähdekoodissa, DataContext varmistaa, että palautettava entiteettiolio on aina sama

entiteettiolion instanssi, mikä haettiin ensimmäisen kerran kyseisen DataContext-luokan instanssin luonnin jälkeen. Jos tietomallissa muutetaan tätä entiteettiolion instanssia, muuttuu se kaikkialla, missä siihen viitataan.

DataContext-luokka toteuttaa tämän toiminnallisuuden tallentamalla entiteettiolion pääavaimen välimuistiin ensimmäisellä kerralla kun entiteettiolio haetaan tietokannasta DataContext-luokan instanssin luomisen jälkeen. Seuraavien hakujen suorituksessa, jotka koskevat samaa entiteettioliota, DataContext-luokka tarkistaa ensiksi, löytyykö vastaavaa entiteettioliota välimuistista, ja jos löytyy, niin se palauttaa välimuistissa olevan entiteettiolion instanssin. [1. s.503.]

Esimerkki 68: Tietomallin entiteettiolioiden identiteetin seuranta.

```
using (DemoDataContext dx = new DemoDataContext())
{
   Product prod1 = dx.Products.First();
   Product prod2 = dx.Products.First();

   Console.WriteLine("prod1: {0}, prod2: {0}",prod1.Name,prod2.Name);
   Console.WriteLine("Changing prod1.Name.");

   prod1.Name = "CHANGED";

   Console.WriteLine("prod1: {0}, prod2: {0}",prod1.Name,prod2.Name);
}
```

Esimerkissä 68 esitellään DataContext-luokan entiteettien identiteetin seurantaa käytännössä. Esimerkissä haetaan kahteen Product-tyyppiseen muuttujaan Products-taulun ensimmäinen tuote, jonka jälkeen tulostetaan molempien muuttujien tuotteen nimi. Tulostuksen jälkeen muutetaan ensimmäisessä muuttujassa olevan tuotteen nimeä ja tulostetaan nimet uudelleen.

Tietomallin muutosten seuranta ja käsittely

Kun DataContext-luokan tietomallin entiteettiolioiden identiteetin seuranta on tallentanut oliot välimuistiin, alkaa olioiden muutosten seuranta. DataContext-luokan entiteettiolioiden muutosten seuranta toimii tallentamalla entiteettiolioiden alkuperäiset arvot välimuistiin. Entiteettiolioiden muutoksia seurataan niin kauan, kunnes kutsutaan DataContext-olion SubmitChanges-metodia, joka tallentaa muutokset tietokantaan.

Entiteettiolioiden identiteetin ja muutosten seuranta toimii automaattisesti niin kauan kuin käsiteltävät oliot haetaan tietokannasta. Entiteettioliota lisättäessä tai poistettaessa pitää DataContext-luokalle ilmoittaa kyseisestä tapahtumasta, että DataContext-olio voi aloittaa kyseisen entiteettiolion identiteetin seurannan ja muutosten seurannan.

Mitään entiteettiolioden lisäys-, poisto- tai päivitysoperaatioita ei suoriteta tietokantaan ennen kuin DataContext-olion SubmitChanges-metodia on kutsuttu, vaan kaikki suoritettavat operaatiot tallennetaan muutosten seurantaan. Kun SubmitChanges-metodia kutsutaan, DataContext-olio aloittaa tietomallin muutosten prosessoinnin, jolloin DataContext-olio luo tietomallin muutosten mukaiset vaadittavat SQL-lauseet ja antaa ne tietokannalle suoritettavaksi. DataContext-olio huolehtii SQL-lauseiden oikeasta suoritusjärjestyksestä, jotta esimerkiksi tietokannan viite-eheys säilyy.

DataContext-olio suorittaa kaikki tietokantaan samanaikaisesti tehtävät muutokset implisiittisesti tietokantatransaktion sisällä, jolloin erillistä transaktioiden hallintaa ei tarvita.

SubmitChanges-metodikutsun jälkeen DataContext-luokan instanssin identiteettiseuranta ja muutosten seuranta palaavat alkutilaan, jossa tietomalli vastaa samoja arvoja kuin tietokanta.

Entiteettiolioiden muutostenseuranta käyttää enemmän resursseja, koska DataContext-luokan on tallennettava välimuistiin haettujen entiteettiolioiden alkuperäiset arvot. Muutostenseuranta voidaan ottaa pois käytöstä, jolloin hakujen suorituskyky lisääntyy, mutta tällöin voidaan vain hakea tietoa.

Muutostenseurannan poistaminen käytöstä on hyödyllistä silloin kun tiedetään etukäteen, että halutaan vain näyttää tietoa tai haetut entiteettioliot annetaan eteenpäin käsiteltäviksi, siten että kyseisen DataContext-luokan instanssi poistuu niiden näkyvyysalueesta, jolloin entiteettiolioiden muutoksia ei voida enää seurata.

Muutostenseurannan voi asettaa pois toiminnasta asettamalla DataContext-luokan ObjectTrackingEnabled-ominaisuus ”falseksi”. Jos muutostenseuranta asetetaan pois toiminnasta sen jälkeen kun kyseisen DataContext-luokan instanssia on jo käytetty tietokantahakuihin, eli instanssi on jo aloittanut entiteettiolioiden muutostenseurannan, aiheutuu InvalidOperationException-poikkeus. Sama poikkeus aiheutuu myös, jos yritetään kutsua SubmitChanges-metodia DataContext-luokan instanssille, jonka muutostenseuranta on otettu pois käytöstä.

7.5 Tietokantaoperaatiot

Tässä luvussa käydään läpi Linq To SQL -tietokantaoperaatiot eli haku-, lisäys-, päivitys- ja poisto-operaatiot. Vastaavista toiminnallisuuksista käytetään yleensä lyhennettä CRUD (Create, Retrieve, Update, Delete) -metodit.

Hakuoperaatiot

Tietokantahakujen muodostus vastaa normaalien hakujen muodostamista standardihakuoperaattoreita käyttämällä.

Hakuoperaatioiden kohteena toimivat DataContext-luokan Table<T>-tyyppiset ominaisuudet, jotka vastaavat tietokannan tauluja. Table<T>-tyyppi toteuttaa IQueryable<T>-rajapinnan, joka periytyy IEnumerable<T>-rajapinnasta, jolloin hakuoperaatioissa voidaan käyttää kaikkia standardihakuoperaattoreita.

Linq To SQL -haut eroavat standardioperaattoreista siinä, että ne palauttava IQueryable<T>-tyyppisiä sekvenssejä IEnumerable<T>-tyyppisten sekvenssien sijaan ja ne muutetaan tietokannalle suoritettaviksi SQL-lauseiksi. Edellisen vuoksi niissä ei voida käyttää olioiden metodeja, koska näitä metodeja ei voida suoraan mallintaa SQL-lauseiksi.

Esimerkki 69: Hakuoperaation suorittaminen esimerkkitietokantaan.

```
using (DemoDataContext dx = new DemoDataContext())
{
    IQueryable<Product> result = dx.Products.Where(p => p.Price < 20);

    foreach (Product p in result)
        Console.WriteLine("{0} - {1}", p.Name, p.Price);
}
```

Esimerkissä 69 haetaan tietokannan Products-taulusta ne tuotteet, joiden hinta on alle 20. Esimerkissä aluksi luodaan DemoDataContext-olio, jonka Products-ominaisuus toimii haun kohteena.

Entiteettiolioiden väliset assosiaatiot

Tietomallissa tietokantataulujen väliset relaatiot mallinnetaan tietomalliin entiteettiolioiden välisinä assosiaatioina, jolloin hakuoperaatioissa ei tarvitse käyttää Join-operaatioita, kun halutaan hakea toisiinsa liittyviä entiteettiolioita.

Esimerkki 70: Entiteettiolioiden välisten assosiaatioiden käyttö tietokantahaussa.

```
using (DemoDataContext dx = new DemoDataContext())
{
    Category cat = dx.Categories
        .SingleOrDefault(c => c.Name == "Books");

    Console.WriteLine(cat.Name);

    foreach (Product p in cat.Products)
        Console.WriteLine("   {0}", p.Name);
}
```

Esimerkissä 70 haetaan ”Books”-niminen kategoria ja listataan sen sisältämät tuotteet. Esimerkissä haetaan Category-olion tuotteet käyttämällä Category-olion ja Products-olion välistä assosiaatiota, joten kategoriaa ja tuotteita ei tarvitse liittää toisiinsa erillisellä liitosoperaatiolla.

Assosiaatioiden väliset viivästetyt ja ei-viivästetyt haut

Linq To SQL käyttää oletuksena viivästettyä tiedonhakua haettaessa entiteettiolion assosioidun jäsenolion tietoja tietokannasta. Entiteettiolioon assosioidun jäsenolion tietoja ei haeta, ennen kuin kyseistä assosiaatiota käytetään, eli entiteettiolion jäsenolioon viitataan lähdekoodissa.

Viivästetty tiedonhaku edellyttää, että DataContext-luokan muutostenseuranta on käytössä, sillä muuten se ei voi seurata, mitä entiteettiolioita on jo haettu tietokannasta.

Esimerkissä 70 esiteltiin haku, jossa haetaan kategorian sisältämät tuotteet käyttämällä Category-luokan Products-assosiaatiota. Esimerkissä suoritetaan itse asiassa kaksi tietokantahakua. Seuraavassa listauksessa on esitetty esimerkin tulostus, kun DataContext-luokan lokitiedot on lisätty tulostukseen:

```
SELECT [t0].[Id], [t0].[Name]
FROM [Category] AS [t0]
WHERE [t0].[Name] = @p0
-- @p0: Input VarChar (Size = 5; Prec = 0; Scale = 0) [Books]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build:
3.5.30729.1

Books
SELECT [t0].[Id], [t0].[Name], [t0].[Price], [t0].[CategoryId]
FROM [Product] AS [t0]
WHERE [t0].[CategoryId] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [2]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build:
3.5.30729.1

   Pro Linq
   Xml
   Linq To Objects
   Professional C#
```

Listauksesta nähdään, että ensimmäinen tietokantahaku tapahtuu ennen kategorian nimen tulostamista eli esimerkin seuraavalla koodirivillä:

```
Category cat = dx.Categories.SingleOrDefault( c => c.Name == "Books");
```

Seuraavaa tietokantahaku suoritetaan, kun käytetään Category-olion Products-assosiaatiota ennen tuotteiden tulostamista seuraavalla koodirivillä:

```
foreach (Product p in cat.Products)
...
```

Viivästetty tiedonhaku on hyödyllinen ominaisuus silloin kun haetaan vain yksittäisiä entiteettiolioita, eli haetaan tietoa vain yhdestä taulusta ja käytetään vain kyseisen olion ominaisuuksia. Tällöin ei turhaan ladata assosioitujen entiteettiolioiden tietoja, jos niitä ei käytetä.

Assosioitujen entiteettiolioiden tietojen hakua voidaan kontrolloida DataContext-luokan DataLoadOptions-tyyppisellä LoadOptions-ominaisuudella. Se sisältää geneerisen LoadWith-metodin, jolla voidaan määrittää, mitkä entiteettiolion jäsenoliot haetaan samaan aikaan kuin kyseisen oliokin. Esimerkissä 71 on esitelty DataContext-luokan LoadOptions-ominaisuuden käyttö.

Esimerkki 71: DataLoadOptions-luokan käyttö entiteettiolioiden tietojenhaun kontrollointiin.

```
using (DemoDataContext dx = new DemoDataContext())
{
    dx.Log = Console.Out;

    // Asetetaan LoadOptions
    DataLoadOptions options = new DataLoadOptions();
    options.LoadWith<Category>(c => c.Products);
    dx.LoadOptions = options;

    Category cat = dx.Categories
        .SingleOrDefault(c => c.Name == "Books");

    Console.WriteLine(cat.Name);
    foreach (Product p in cat.Products.Where(p => p.Price > 10))
        Console.WriteLine("   {0}", p.Name);
}
```

Esimerkissä suoritetaan sama haku kuin esimerkissä Esimerkki 70, jossa haetaan kategorian tuotteet käyttämällä assosiaatiota. Esimerkissä asetetaan kategorian tuotteiden tietojenhaku välittömäksi käyttämällä DataLoadOptions-oliota. Ladattava jäsenolio määritetään LoadWith-metodille annettavalla delegaattina. Esimerkki tulostaa seuraavaa:

```
SELECT [t0].[Id], [t0].[Name], [t1].[Id] AS [Id2], [t1].[Name] AS [Name2],
[t1].[Price], [t1].[CategoryId], (
    SELECT COUNT(*)
    FROM [Product] AS [t2]
    WHERE [t2].[CategoryId] = [t0].[Id]
    ) AS [value]
FROM [Category] AS [t0]
LEFT OUTER JOIN [Product] AS [t1] ON [t1].[CategoryId] = [t0].[Id]
WHERE [t0].[Name] = @p0
ORDER BY [t0].[Id], [t1].[Id]
-- @p0: Input VarChar (Size = 5; Prec = 0; Scale = 0) [Books]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build:
3.5.30729.1

Books
   Pro Linq
   Xml
   Linq To Objects
   Professional C#
```

Esimerkin tulosteesta nähdään, että nyt suoritetaan tietokantaan vain yksi SQL-haku, koska Category-luokan Products-ominaisuus on asetettu välittömästi ladattavaksi.

Viivästetty tiedonhaku voidaan ottaa pois käytöstä DataContext-luokan instanssilta asettamalla sen DeferredLoadingEnabled-ominaisuus pois toiminnasta. Tällöin 1-1-suhteessa olevien entiteettiolioiden assosiaatiot palauttavat aina arvon null ja 1-N-suhteessa olevat assosiaatiot palauttavat tyhjän kokoelman. Vaikka viivästetty tiedonhaku otetaan pois käytöstä, voidaan yhä ladata entiteettiolioiden jäsenolioita käyttämällä LoadOptions-ominaisuutta.

Lisäysoperaatiot

Entiteettiolion lisäysoperaatio suoritetaan luomalla uusi instanssi oliosta ja lisäämällä se joko DataContext-olion vastaavaan Table<T>-tyyppiseen tauluun tai liittämällä entiteettiolion instanssi jo aiemmin haetun entiteettiolion EntitySet<T>-kokoelmaan, jolloin DataContext-luokka saa tiedon lisätystä entiteettioliosta ja pystyy muodostamaan tarvittavat SQL INSERT -lausekkeet kun DataContext-olion SubmitChanges-metodia kutsutaan.

Entiteettioliot liitetään DataContext Table<T>-tyyppiseen tauluun kutsumalla kyseisen ominaisuuden InsertOnSubmit<T>-metodia, jossa T on lisättävän entiteettiolion tyyppi.

DataContext-luokka osaa seurata lisättävän entiteetiolion assosiaatioita, jolloin lisättävän entiteettiolion assosioidut jäsenoliot lisätään myös tietokantaan samalla kun kyseinen entiteettiolio lisätään.

Esimerkki 72: Entiteettiolioiden lisääminen tietokantaan.

```
// 1. Instantiodaan DataContext
using (DemoDataContext dx = new DemoDataContext())
{
    // 2. Instantioidaan entiteettioliot
    Category cat = new Category
    {
        Name = "TempCategory",
        Products = {
            new Product { Name = "Product #1", Price = 10 },
            new Product { Name = "Product #2", Price = 20 }
         }
    };

    // 3. Lisätään entiteettiolio DataContext Table<T> kokoelmaan
    dx.Categories.InsertOnSubmit(cat);

    // 4. Kutsutaan DataContext SubmitChanges metodia
    dx.SubmitChanges();

    // Haetaan lisätty olio
    Console.WriteLine("ID {0} : {1}", cat.Id, cat.Name);

    foreach (Product p in cat.Products)
        Console.WriteLine("   ID {0} : {1} - {2}",
                p.Id, p.Name, p.Price);
}
```

Esimerkissä 72 esitellään entiteettiolion lisäysoperaatio tietokantaan. Esimerkissä lisätään tietokantaan uusi kategoria ja sille kaksi tuotetta. Esimerkin alussa luodaan DemoDataContext-luokan instanssi ja lisättävien entiteettiolioiden instanssit. Esimerkin kohdassa 3 liitetään luotu Category-entiteettiolio tietomalliin kutsumalla DemoDataContext-olion Categories-ominaisuuden InsertOnSubmit-metodia. Lopuksi kutsutaan DemoDataContext-olion SubmitChanges-metodia, jolloin tietomallin muutokset päivitetään tietokantaan. Esimerkin lopussa tulostetaan juuri lisätyn kategorian tiedot.

Päivitysoperaatiot

Tietomallin entiteettiolioiden päivitysoperaatiot tapahtuvat muokkaamalla tietomallin entiteettioliota ja kutsumalla DataContext-luokan SubmitChanges-metodia, jolloin tietomallin muutokset päivitetään tietokantaan.

Esimerkki 73: Entiteettiolion päivittäminen.

```
using (DemoDataContext dx = new DemoDataContext())
{
    IQueryable<Product> cars = dx.Products
        .Where(p => p.Category.Name == "Cars");
    foreach (Product p in cars)
        Console.WriteLine("Name: {0} - Price: {1}",p.Name,p.Price);

    // Muokataan entiteettioliota
    Product ferrari = cars.Single(p => p.Name == "Ferrari");
    ferrari.Price = 300000;

    // Päivitetään tietomallin muutokset
    dx.SubmitChanges();
    Console.WriteLine("---");

    foreach (Product p in cars)
        Console.WriteLine("Name: {0} - Price: {1}",p.Name,p.Price);
}
```

Esimerkissä 73 esitellään entiteettiolion päivitysoperaatio. Esimerkissä haetaan ”Cars”-kategorian ”Ferrari”-niminen tuote ja muutetaan sen hinnaksi 300 000. Tietomallin muutokset päivitetään tietokantaan kutsumalla DemoDataContext-olion SubmitChanges-metodia. Esimerkissä listataan ”Cars”-kategorian tuotteet ennen tietomallin päivitystä ja tietomallin päivityksen jälkeen.

Entiteettiolioiden välisten assosiaatioiden päivittäminen tapahtuu muokkaamalla entiteettiolioiden EntityRef<T>- tai EntitySet<T>-tyyppisiä ominaisuuksia ja kutsumalla DataContext-luokan SubmitChanges-metodia. DataContext osaa seurata entiteettiolioiden välisten assosiaatioiden muutoksia ja tehdä vastaavat viiteavainten päivitykset tietokantaan.

Äitiolion lapsiolioiden referenssit voidaan päivittää lisäämällä lapsiolio äitiolion EntitySet<T>-tyyppiseen kokoelmaan käyttämällä kokoelman Add-metodia tai poistamalla lapsiolio kokoelmasta kokoelman Remove-metodilla.

Esimerkki 74: Äitiolion lapsiolioiden referenssien päivittäminen lisäämällä lapsiolio äitiolion EntitySet<T> kokoelmaan.

```
using (DemoDataContext dx = new DemoDataContext())
{
    Category books = dx.Categories.Single(p => p.Name == "Books");
    Product ferrari = dx.Products.Single(p => p.Name == "Ferrari");

    books.Products.Add(ferrari);

    dx.SubmitChanges();
}
```

Esimerkissä 74 esitellään entiteettiolioiden välisten assosiaatioiden päivittäminen lisäämällä lapsiolio äitiolion EntitySet<T>-kokoelmaan. Esimerkissä päivitetään ”Ferrari”-niminen tuote kuulumaan ”Books”-nimiseen kategoriaan lisäämällä se Category-entiteettiolion Products-kokoelmaan.

Lapsiolioiden referenssi äitiolioon voidaan päivittää asettamalla lapsiolion EntityRef<T>-tyyppinen ominaisuus viittaamaan toiseen entiteettiolioon.

Esimerkki 75: Lapsiolion äitiolion referenssin päivittäminen.

```
using (DemoDataContext dx = new DemoDataContext())
{
    Product ferrari = dx.Products.Single(p => p.Name == "Ferrari");
    Category cars = dx.Categories.Single(c => c.Name == "Cars");

    ferrari.Category = cars;

    dx.SubmitChanges();
}
```

Esimerkissä 75 esitellään lapsiolion äitiolion referenssin päivittäminen. Esimerkissä asetetaan ”Ferrari”-niminen tuote kuulumaan ”Cars”-nimiseen kategoriaan asettamalla sen Category-ominaisuus viittaamaan cars-entiteettiolioon.

Poisto-operaatiot

Entiteettiolioiden poistaminen tietomallista tapahtuu kutsumalla DataContext-luokan vastaavan Table<T>-tyyppisen ominaisuuden DeleteOnSubmit-metodia, joka merkitsee entiteettiolion poistettavaksi tietomallista seuraavan päivityksen yhteydessä ja päivittämällä tietomallin muutokset tietokantaan kutsumalla SubmitChanges-metodia.

Poistettaessa entiteettioliota, jolla on lapsiolioita ja eikä tietomalli salli kyseisen entiteettiolion poistamista, ellei lapsiolioita poisteta myös, pitää entiteettiolion lapsioliot merkitä poistettavaksi kutsumalla niille vastaavasti DeleteOnSubmit-metodia tai DeleteAllOnSubmit-metodia, jolla voidaan merkitä kokoelma entiteettioliota poistettavaksi yhdellä kertaa.

Esimerkki 76: Entiteettiolion ja lapsiolioiden poistaminen.

```
using (DemoDataContext dx = new DemoDataContext())
{
    Category cat = dx.Categories
        .SingleOrDefault(c => c.Name == "TempCategory");

    if (cat != null)
    {
        dx.Categories.DeleteOnSubmit(cat);

        // Poistetaan lapsioliot, jos niitä on
        if (cat.Products.Any())
            dx.Products.DeleteAllOnSubmit(cat.Products);

        dx.SubmitChanges();
    }
}
```

Esimerkissä 76 esitellään entiteettiolion ja sen lapsiolioiden poisto-operaatio. Esimerkissä poistetaan ”TempCategory”-niminen kategoria ja sen tuotteet. Poistettava kategoria haetaan

ensiksi tietokannasta ja se merkitään poistettavaksi. Seuraavaksi tarkistetaan Any-standardihakuoperaattorilla, onko kyseisessä kategoriassa tuotteita, ja jos kategoriassa on tuotteita, merkitään ne kaikki myös poistettaviksi. Lopuksi päivitetään tietomallin muutokset tietokantaan.

7.6 Tietomallin toiminnallisuuden laajentaminen

Tietomallin DataContext-luokasta perityn luokan ja entiteettiluokkien lähdekoodin generoinnin yhteydessä luokkiin generoidaan joukko partial-metodeja, joilla voidaan laajentaa ja muokata niiden toiminnallisuutta.

DataContext-luokan partial-metodit

DataContext-luokasta perittyyn luokkaan generoidaan seuraavat partial-metodit:

- OnCreated
- Insert_(TEntity instance)
- Update_(TEntity instance)
- Delete_(Tentity instance).

OnCreated-metodia kutsutaan DataContext-luokan rakentimesta, joten se mahdollistaa DataContext-luokan rakentimen toiminnallisuuden laajentamisen.

Luokkaan generoidaan jokaista entiteettiolion tietokantaoperaatiota vastaava partial-metodi: Insert_, Update_ ja Delete_, jossa ”_” vastaa entiteettiluokan nimeä, joita kutsutaan silloin kun suoritetaan kyseinen tietokantaoperaatio entiteettioliolle. Metodit saavat parametrikseen TEntity-tyyppisen entiteettiolion instanssin, johon kyseinen tietokantaoperaatio kohdistuu, ja ne mahdollistavat tietokantaan suoritettavan tietokantaoperaation muokkaamisen. Esimerkiksi entiteettiolion lisäysoperaatiossa voidaan käyttää omaa tietokantaproseduuria Linq To SQL -suoritusvaiheen muodostaman dynaamisen SQL-lauseen sijaan.

DataContext-luokka sisältää seuraavat metodit: ExecuteDynamicInsert, ExecuteDynamicUpdate ja ExecuteDynamicDelete, joilla voidaan muodostaa dynaamisesti vaadittavat SQL-lausekkeet entiteettiluokan instanssista. Esimerkiksi ExecuteDynamicUpdate muodostaa tarvittavat SQL-lausekkeet entiteettiluokan päivittämiseksi.

Esimerkki 77: DemoDataContext-luokan DeleteCategory partial-metodin toteutus.

```
partial class DemoDataContext
{
    partial void DeleteCategory(Category instance)
    {
        if(instance.Products.Any())
        {
            foreach(Product p in instance.Products)
                this.ExecuteDynamicDelete(p);
        }

        this.ExecuteDynamicDelete(instance);
    }
}
```

Esimerkissä 77 on esitelty DemoDataContext-luokan DeleteCategory partial-metodin toteutus. Kyseistä metodia kutsutaan silloin kun Category-tyyppinen entiteettiolio on asetettu poistettavaksi ja muutokset päivitetään tietokantana. Esimerkkitoteutuksessa tarkistetaan, kuuluuko poistettavaan kategoriaan tuotteita, ja jos kuuluu, muodostetaan myös näille tuotteille SQL DELETE -lauseet ExecuteDynamicDelete-metodilla.

Koska esimerkin DeleteCategory partial-metodin toteutus huolehtii nyt kategoriaan kuuluvien tuotteiden poistosta, ei ohjelmakoodissa enää tarvitse kategoriaa poistettaessa erikseen asettaa kategorian tuotteita poistettavaksi.

DataContext-luokan partial-metodien toteuttamisessa vastuu tietokantaoperaatioiden toimivasta toteuttamisesta siirtyy ohjelmoijalle, koska tällöin DataContext-luokka käyttää metodien oletustoteutuksien sijaan ohjelmoijan toteuttamia partial-metodeja tietokantaoperaatioiden suorittamiseen. Tällöin niiden toteutuksessa noudatettava seuraavia sääntöjä [1. s. 449–453; 15]:

- Metodin toteutuksesta ei voida kutsua SubmitChanges- tai Attach-metodeita, muuten aiheutuu poikkeus.
- Metodit eivät voi aloittaa uutta, suorittaa tai lopettaa tietokantatransaktiota, sillä SubmitChanges-operaatio suoritetaan transaktion sisällä.
- Metodien toteutuksien oletetaan noudattavan Linq To Sql:n käyttämää optimistista samanaikaisuuden hallintaa. Konfliktien sattuessa metodien oletetaan aiheuttavan ChangeConflictException-poikkeuksen, joka voidaan käsitellä oikein SubmitChanges-metodin suorituksen yhteydessä.
- Insert_ ja Update_ partial-metodien toteutuksen pitää palauttaa tietokannassa generoidut arvot oikein, kuten esimerkiksi IDENTITY-kenttien arvot.

- Ohjelmoijan vastuulla on kutsua oikeata dynaamista tietokantaoperaatiota. Jos esimerkiksi kutsutaan ExecuteDynamicInsert-metodia partial-metodin toteutuksesta, jonka pitäisi suorittaa päivitysoperaatio, on lopputulos epämääräinen.

Tietokantaoperaatioiden ylikirjoittamisessa on syytä noudattaa huolellisuutta, eritoten samanaikaisuuden hallinnan ja konfliktien havaitsemisen yhteydessä, muuten tietokantaoperaatioiden toiminta saattaa muuttua epämääräiseksi tai täysin toimimattomaksi.

Entiteettiluokkien partial-metodit

Entiteettiluokkiin generoidaan seuraavat partial-metodit:

- OnCreated()
- OnLoaded()
- OnValidate(ChangeAction action)
- On_Changing (T value)
- On_Changed().

OnCreated partial -metodia kutsutaan entiteettiluokan rakentimesta, joten jos halutaan muokata esimerkiksi alkuarvojen alustamista, niin voidaan toteuttaa tämä metodi.

OnLoaded partial -metodia kutsutaan, kun kyseinen entiteettiolio ladataan tietokannasta.

OnValidate(ChangeAction action) partial -metodia kutsutaan tietomallin päivityksen yhteydessä, silloin kun entiteettiolio kuuluu tietokantaoperaation suoritukseen. Metodin päätarkoitus on tarkistaa olion tietojen oikeellisuus ennen kuin tiedot tallennetaan tietokantaan. Metodi saa parametrikseen ChangeAction-tyyppisen enumeraation, joka kertoo, minkä tyyppisestä tietokantaoperaation suorituksesta on kyse.

Esimerkki 78: Category-entiteettiluokan OnValidate-partial metodin toteutus.

```
partial class Category
{
    partial void OnValidate(ChangeAction action)
    {
        if(action == ChangeAction.Insert)
        {
            if (!this.Products.Any())
                throw new Exception("Can’t insert empty category");
        }
    }
}
```

Esimerkissä 78 on toteutettu Category-entiteettiluokalle OnValidate partial-metodi, jossa tehdään tarkistus, että tietokantaan lisättävällä kategorialla on myös tuotteita. Jos lisättävässä kategoriassa ei ole tuotteita, aiheutetaan poikkeus.

Jokaista entiteettiolion ominaisuutta kohden generoidaan On_Changing(T value)- ja On_Changed partial -metodit, joissa ”_” vastaa ominaisuuden nimeä. Näitä metodeita kutsutaan ominaisuuden asetuslohkosta, ja niillä on mahdollista laajentaa sen asetuslogiikkaa. On_Changing(T value)-metodia kutsutaan ennen kuin entiteettiolion jäsenmuuttujan arvo asetetaan ja On_Changed-metodia kutsutaan arvon asettamisen jälkeen. On_Changing-metodi saa value-parametrina kyseiseen ominaisuuteen sijoitettavan uuden arvon.

Esimerkki 79: Product-entiteettiluokan OnPriceChanging partial-metodin toteutus.

```
partial class Product
{
    partial void OnPriceChanging(decimal value)
    {
        if(value <= 0)
            throw new ArgumentException ("Invalid value");
    }
}
```

Esimerkissä 79 on esitelty Product-entiteettiluokan OnPriceChanging partial -metodin toteutus, jossa tarkistetaan Price-ominaisuuteen sijoitettavan arvon oikeellisuus. Jos sijoitettava arvo on nolla tai alle, aiheutetaan poikkeus.

OnValidate partial-metodi muodostaa yhdessä On_Changing-metodien kanssa loogisen paikan sijoittaa tietomallin bisneslogiikka ja tarkistuslogiikka.

7.7 Samanaikaisuuden hallinta ja konfliktien ratkaisu

Samanaikaisuuden hallinnalla tarkoitetaan sitä, että kun samaa tietokantaa käyttää usea tietokantayhteys samaan aikaan, voi jossain vaiheessa kaksi tietokantayhteyttä yrittää muokata samaa tietoa, jolloin syntyy konflikti.

Esimerkiksi tietokantayhteys A lukee tiedon, jonka jälkeen tietokantayhteys B käy lukemassa saman tiedon, jonka jälkeen A käy päivittämässä tiedon. Tällöin B:n aikaisemmin lukema tieto ei enää vastaa tietokannassa olevaa tietoa. Kun B yrittää päivittää tietoa, syntyy konflikti, koska B:n päivittämä tieto perustuu vanhaan, ei enää voimassa olevaan tietoon.

Konflikti ratkaisulla tarkoitetaan sitä, että päätellään jollain logiikalla, mikä konfliktissa oleva tieto päivitetään tietokantaan. Ratkaisulogiikka riippuu yleensä sovelluksen käsittelemästä tiedosta.

Samanaikaisten konfliktien hallintaan on olemassa useampi toteutusmalli. Pessimistinen konfliktienhallinta lukitsee tietokannan tiedon, siten että tietoa voi käsitellä vain yksi tietokantayhteys kerrallaan, tällöin konflikteja ei pääse syntymään, mutta tämä malli aiheuttaa suurella yhtäaikaisella käyttäjämäärällä usein tarpeettomasti tietokannan lukkojen aukeamisen odottamista eikä sovellu juurikaan suurelle käyttäjämäärälle.

Optimistinen konfliktien hallinta ei lukitse tietokannan tietoja, vaan se perustuu siihen oletukseen, että konflikteja syntyy vain harvoin. Optimistinen konfliktien hallinta on huomattavasti monimutkaisempi toteuttaa kuin pessimistinen hallinta, sillä on toteutettava oma logiikka siihen, miten konflikti havaitaan ja ratkaistaan.

Linq To SQL käyttää optimistista konfliktienhallintaa.

Konfliktien havaitseminen

Konfliktien havaitseminen on toteutettu siten, että tietomalliin määritellään entiteettiolioiden ominaisuuksiin attribuuteilla, mitä ominaisuuksia käytetään konfliktin havaitsemiseen. Kun tietokantaa ollaan päivittämässä, dynaamisesti muodostettavaan SQL UPDATE -lauseen ehtolausekkeeseen sijoitetaan päivitettävän entiteettiolion niiden ominaisuuksien alkuperäiset arvot, joita käytetään konfliktin havaitsemiseen. Jos suoritettava SQL UPDATE -lause ei tällöin löydä annetuilla ehdoilla päivitettävää riviä tietokannasta, tiedetään, että ehdossa käytetyt alkuperäiset arvot eivät enää vastaa tietokannassa olevia arvoja, joten voidaan todeta konflikti.

Konfliktin tapahtuessa DataContext-luokan SubmitChanges-metodi aiheuttaa ChangeConflictException-poikkeuksen.

Esimerkki 80: Product-entiteettiolion päivitysoperaation dynaamisesti muodostettu SQL UPDATE -lause.

```
UPDATE [Product]
SET [Price] = @p4
WHERE
    ([Id] = @p0)
    AND ([Name] = @p1)
    AND ([Price] = @p2)
    AND ([CategoryId] = @p3)
```

Esimerkissä 80 esitellään tietokantaan suoritettava SQL UPDATE -lauseke, joka on dynaamisesti muodostettu Product-entiteettiolion päivitysoperaatiossa. Esimerkin

lausekkeesta nähdään, että WHERE-ehtolausekkeessa ovat mukana kaikki Product-entiteettiolion ominaisuudet, joita käytetään konfliktien havaitsemiseen.

Ne entiteettiolioiden ominaisuudet, joita käytetään konfliktin havaitsemiseen, määritellään tietomalliin ColumnAttribute-attribuutin IsVersion- ja UpdateCheck-parametreilla.

Jos entiteettiolion jollekin ominaisuudelle on asetettu ColumnAttribute-attribuutin parametri IsVersion=true, konfliktien havaitsemiseen käytetään pelkästään tätä ominaisuutta. Tällöin kyseinen entiteettiolion ominaisuus toimii versionumerona, jota kasvatetaan aina, kun päivitysoperaatio suoritetaan. Versionumeron tiedon tyyppinä voi toimia SQL Server -tietokannan TimeStamp-tyyppi tai IDENTITY-arvo, jota tietokanta automaattisesti kasvattaa. IsVersion-parametrin oletusarvo on false.

Esimerkki 81: Product-entiteettiolion Version-ominaisuuden määrittely.

```
[Column(Storage="_Version",
 AutoSync=AutoSync.Always,
 DbType="timestamp",
 CanBeNull=false,
 IsDbGenerated=true,
 IsVersion=true)]
private System.Data.Linq.Binary Version
{
    ...
}
```

Esimerkissä 81 on esitelty, miten entiteettioliolle voidaan määritellä versionumero-ominaisuus käyttämällä ColumnAttribute-attribuuttia ja sen IsVersion-parametria.

Esimerkissä 82 on esitelty dynaamisesti muodostettu SQL UPDATE -lauseke Product-entiteettiolion päivityksestä, jossa käytetään versionumero-ominaisuutta konfliktin havaitsemiseen. Esimerkistä nähdään, että nyt ei käytetä konfliktin havaitsemiseen muuta tietoa kuin Version-ominaisuuden arvoa. Esimerkin SQL UPDATE -lauseen jälkeen suoritetaan myös SELECT-kysely, jolla haetaan Versio-ominaisuuteen uusi arvo.

Esimerkki 82: Product-entiteettiolion dynaamisesti muodostettu SQL UPDATE -lause, jossa käytetään versionumero-ominaisuutta konfliktin havaitsemiseen.

```
UPDATE [Product]
SET [Price] = @p2
WHERE ([Id] = @p0) AND ([Version] = @p1)

SELECT [t1].[Version]
FROM [Product] AS [t1]
WHERE ((@@ROWCOUNT) > 0) AND ([t1].[Id] = @p3)
```

ColumnAttribute-attribuutin UpdateCheck-parametrilla on mahdollista tarkemmin määrittää, mitä kaikkia entiteettiluokan ominaisuuksia käytetään konfliktin havaitsemiseen ja milloin niitä käytetään. UpdateCheck-parametrille voidaan antaa seuraavat arvot:

- Allways – ominaisuutta käytetään aina konfliktin havaitsemiseen.
- Newer – ominaisuutta ei koskaan käytetä konfliktin havaitsemiseen.
- WhenChanged – ominaisuutta käytetään konfliktin havaitsemiseen vain silloin kun sen arvo on muuttunut.

UpdateCheck-parametrin oletusarvo on AllWays, joten Linq To SQL käyttää oletuksena konfliktien havaitsemiseen kaikkia entiteettiluokan ominaisuuksia. Tämä saattaa aiheuttaa pitkien SQL-kyselyiden muodostusta ja haitata näin olleen suorituskykyä. On suositeltavaa, että tietomallia luotaessa otetaan pois käytöstä konfliktien havaitseminen niiltä entiteettiluokkien ominaisuuksilta, joilta sitä ei tarvita, tai vaihtoehtoisesti käytetään versionumeroiden tarkistusta.

Konfliktien ratkaiseminen

Samanaikaisten konfliktien havaitseminen tapahtuu DataContext-luokan SubmitChanges-metodia kutsuttaessa. Jos suoritettavassa transaktiossa tapahtuu konflikti, metodi heittää ChangeConflictException-poikkeuksen ja koko transaktio palautetaan takaisin alkutilaan, mikä merkitsee myös sitä, että ne päivitysoperaatiot, jotka onnistuivat oikein, palautetaan myös.

Konfliktin havaittua SubmitChanges-metodi tallentaa entiteettioliokohtaisen tiedon konfliktista DataContext-luokan ChangeConflicts-kokoelmaan. Kokoelmaan tallennetaan tietoja konfliktissa olevasta entiteettiolion ominaisuudesta, kuten esimerkiksi kyseisen kentän arvo tietokannassa, ominaisuuden nykyinen ja alkuperäinen arvo. Näiden tietojen avulla konflikti pystytään ratkaisemaan halutulla tavalla.

SubmitChanges-metodille voidaan antaa lisäparametrina ConflictMode-tyyppinen enumeraatio, jolla voidaan määrittää, missä vaiheessa tietokantaoperaatioiden suorittamista metodi aiheuttaa poikkeuksen konfliktin sattuessa. Enumeraation FailOnFirstConflict-arvolla poikkeus aiheutetaan ensimmäisen konfliktin sattuessa ja ContinueOnConflict-arvolla kaikki tietokantaoperaatiot suoritetaan loppuun asti, jonka jälkeen aiheutetaan poikkeus, jos operaatioiden suorituksessa ilmenee konflikteja. Tällöin kaikki konfliktit voidaan käsitellä samalla kertaa.

Konfliktien ratkaiseminen tapahtuu ottamalla kiinni ChangeConflictException-poikkeus, ratkaisemalla konfliktit halutulla tavalla ja kutsumalla uudelleen SubmitChanges-metodia.

Konfliktin ratkaisu tapahtuu kutsumalla ChangeConflictCollection-luokan ResolveAll-metodia tai ObjectChangeConflict- ja MemberChangeConflict-luokkien Resolve-metodeja. Metodit saavat parametrikseen RefreshMode-enumeraation, joka määrittää, miten konflikti ratkaistaan. Enumeraatiolla on seuraavat arvot:

- KeepCurrentValues – kaikki tietokannassa olevat arvot ylikirjoitetaan nykyisillä arvoilla.
- KeepChanges – vain muuttuneet arvot ylikirjoitetaan tietokantaan, eli tietokannan arvot nykyiset arvot yhdistetään.
- OverwriteCurrentValues – säilytetään tietokannassa olevat arvot.

Metodit käyttävät oletuksena KeepCurrentValues-arvoa.

Linq To SQL mahdollistaa konfliktien ratkaisemisen kolmella eri tavalla:

1. ratkaisemalla kaikki konfliktit automaattisesti kutsumalla ChangeConflictCollection-kokoelman ResolveAll-metodia
2. ratkaisemalla konfliktit entiteettioliokohtaisesti käyttämällä ObjectChangeConflict-luokan Resolve-metodia
3. ratkaisemalla konfliktit entiteettiolion ominaisuuskohtaisesti käyttämällä MemberChangeConflict-luokan Resolve-metodia.

Esimerkissä 83 on esitelty kaikkien konfliktien ratkaiseminen automaattisesti ylikirjoittamalla tietokannan arvot. Esimerkissä ResolveAll-metodille annettu true-parametri tarkoittaa, sitä että myös poistettujen rivien konfliktit ratkaistaan automaattisesti.

Esimerkki 83: Kaikkien konfliktien ratkaisu automaattisesti ylikirjoittamalla tietokannan arvot.

```
using (DemoDataContext dx = new DemoDataContext())
{
  // muutetaan tietomallia
  try
  {
    dx.SubmitChanges(ConflictMode.ContinueOnConflict);
  }
  catch (ChangeConflictException)
  {
    // Konflikteja tapahtui
    // Ratkaistaan kaikki konfliktit ylikirjoittamalla
    // tietokannan arvot
    dx.ChangeConflicts
          .ResolveAll(RefreshMode.KeepCurrentValues, true);
    dx.SubmitChanges();
  }
}
```

Esimerkissä 84 on esitelty monimutkaisempi konfliktien ratkaisemistapa, jossa tarkistetaan, onko Product-entiteettiolion Price-ominaisuuden päivittämisessä tapahtunut konflikti. Kyseisen konfliktin ratkaisussa säilytetään tietokannan arvo. Kaikki muut konfliktit ratkaistaan siten, että ylikirjoitetaan tietokannan arvot.

Esimerkki 84: Product-entiteettiluokan Price-ominaisuuden konfliktin ratkaiseminen säilyttämällä tietokannan arvo.

```
using (DemoDataContext dx = new DemoDataContext())
{
   // muutetaan tietomallia

   Product p = dx.Products.First();
   p.Name = "BMV - Conflict";
   p.Price = 150000;

   try
   {
      dx.SubmitChanges(ConflictMode.ContinueOnConflict);
   }
   catch (ChangeConflictException)
   {
      // Konflikteja tapahtui
      foreach (ObjectChangeConflict objConflict
         in dx.ChangeConflicts)
      {
         if (objConflict.Object is Product)
         {
            foreach (MemberChangeConflict memConflict
               in objConflict.MemberConflicts)
            {
               if (memConflict.Member.Name == "Price") {
                  memConflict
                     .Resolve(RefreshMode.OverwriteCurrentValues);
               }else{
                  memConflict
                     .Resolve(RefreshMode.KeepCurrentValues);
               }
            }
         }else{
            objConflict.Resolve(RefreshMode.KeepCurrentValues);
         }
      }
      dx.SubmitChanges();
   }
}
```

Esimerkin konfliktin ratkonnassa käydään nyt läpi kaikki DataContext-luokan ChangeConflicts-kokoelman ObjectChangeConflict-oliot. Jos konfliktiolion entiteettiolio on tyyppiä Product, käydään läpi kaikki sen konfliktissa olevat ominaisuudet. Jos konfliktissa olevan ominaisuuden nimi on ”Price”, konflikti ratkaistaan Resolve-metodilla antamalla parametriksi OverwriteCurrentValues-arvo, jolloin tietokannassa oleva arvo säilyy.

7.8 Linq To SQL:n käyttö N-tasoarkkitehtuurissa

N-tasoarkkitehtuurilla tarkoitetaan ohjelmistoarkkitehtuuria, jossa tiedon esittämislogiikka, tiedon prosessointilogiikka ja tiedon tallennuslogiikka erotellaan erillisiksi tasoiksi. Tasot ovat toisistaan irrallisia, eli esimerkiksi tallennuslogiikkatason toteutusta voidaan muuttaa ilman, että prosessointilogiikkatasoa joudutaan tekemään myös muutoksia.

Web-sovellukset tai SOA (Service Oriented Architecture) käyttävät löyhästi sidottua N-tasoarkkitehtuuria, jossa palvelin ja asiakas välittävät tietoa siten, että palvelin ei suoraan seuraa tiedon tilaa. Usein palvelin myös palvelee samaan aikaan useita asiakkaita, jolloin samanaikaisuuden hallinta ja konfliktien havaitsemisen tärkeys myös kasvaa.

N-tasoarkkitehtuurissa oleellista on tiedon välittäminen tasolta toiselle, mihin yleensä käytetään DTO (Data Transfer Object) -olioita. DTO-oliot kapseloivat välitettävän tiedon luokkiin, jolloin tasojen väliset rajapinnat yksinkertaistuvat. Entiteettiolio voi toimia DTO-oliona. Esimerkiksi asiakasohjelma voi pyytää palvelimelta tietyn entiteettiolion, jonka tiedot asiakasohjelma päivittää ja lähettää päivitetyn entiteettiolion takaisin palvelimelle, jonka palvelin sitten tallentaa tietokantaan.

Kun entiteettiolio serialisoidaan sovellustasolta toiselle, se poistuu DataContext-luokan instanssin näkyvyysalueelta ja muutostenseurannasta, eli siitä tulee niin sanotusti irrallinen ja sen päivitys ja poistaminen ei enää onnistu normaalilla tavalla. Jotta irrallisten entiteettiolioiden muutoksia voitaisiin päivittää, pitää ne liittää takaisin tietomallin muutostenseurantaan ITable-rajapinnan Attach- ja AttachAll-metodeilla, jonka jälkeen tietomallin muutokset voidaan päivittää normaalisti.

Sovelluksesta riippuva samanaikaisuuden hallintamenetelmä vaikuttaa siihen, miten entiteettioliot liitetään takaisin tietomallin muutostenseurantaan ja mitä entiteettiolioiden tietoja pitää välittää sovelluksen tasojen välillä.

Konfliktienhallinnassa voidaan käyttää entiteettiolion versionumero-ominaisuutta tai alkuperäisiä arvoja konfliktien havaitsemiseen. Konfliktien havaitsemiseen käytettävät alkuperäiset arvot pitää välittää DataContext-luokalle irrallista entiteettiolioita liitettäessä. Tällöin esimerkiksi palvelinohjelman tai asiakasohjelman on tallennettava alkuperäiset arvot välimuistiin, jotta ne voidaan välittää yhdessä muuttuneiden arvojen kanssa DataContext-luokalle.

Attach-metodi saa aina parametrikseen liitettävän entiteettiolion, jonka se liittää tietomallin muutostenseurantaan ”muuttumaton” tilassa. Metodille voidaan myös antaa asModified-

lisäparametrina tieto siitä, missä tilassa entiteettiolio liitetään, tai muuttuneen entiteettiolion alkuperäiset arvot. AttachAll-metodilla voidaan liittää sekvenssi olioita kerralla.

Entiteettioliota liitettäessä Attach- ja AttachAll-metodit aiheuttavat InvalidOperationException-poikkeuksen, jos konfliktien havaitsemiseen käytettäviä entiteettiolion ominaisuuksia ei ole asetettu oikein.

Seuraavissa esimerkeissä esitellään eri tapoja päivittää entiteettioliota prosessointilogiikkatasolla. Prosessointilogiikkatasoa kuvaa ProductService-luokka, jonka metodit toteuttavat Product-entiteettiluokan haku-, päivitys- ja poisto-operaatioita.

Esimerkki 85: Entiteettiolion päivittäminen käyttäen versionumero-ominaisuutta konfliktien havaitsemiseen.

```
public Product UpdateProduct(Product product)
{
    using (DemoDataContext dx = new DemoDataContext())
    {
        // Liitetään entiteettiolio muutostenseurantaan
        dx.Products.Attach(product, true);
        CommitChanges(dx);
    }
    return product;
}
```

Esimerkissä 85 esitellään ProductService-luokan UpdateProduct-metodi, jossa suoritetaan Product-entiteettiolion päivittäminen käyttämällä versionumero-ominaisuutta. Päivitettävä entiteettiolio liitetään tietomallin muutostenseurantaan Attach-metodilla ”muuttunut”-tilassa ja kutsutaan ProductService-luokan sisäistä CommitChanges-apumetodia, joka suorittaa tietomallin päivityksen ja konfliktien käsittelyn.

Esimerkissä 86 on esitelty, miten asiakasohjelma voisi käyttää ProductService-luokan UpdateProduct-metodia entiteettiolion päivittämiseen.

Esimerkki 86: Asiakasohjelman, jossa entiteettiolion päivittämisessä käytetään versionumeroa.

```
// Haetaan entiteettiolio
ProductService service = new ProductService();
Product product = service.GetProductById(1);

// Päivitetään entiteettiolion tietoja
product.Name = "BMV Mark II.";
product.Price = 350000;

// Lähetetään enttiteettiolio pävitettäväksi tietokantaan
product = service.UpdateProduct(product);
```

Esimerkissä 87 on esitelty ProductService-luokan toinen versio UpdateProduct-metodista, jolle välitetään alkuperäiset arvot ja muuttuneet arvot. Esimerkissä liitetään Attach-metodilla nyt sekä muuttunut entiteettiolio ja alkuperäinen entiteettiolio muutostenseurantaan.

Esimerkki 87: Entiteettiolion päivittäminen käyttäen alkuperäisiä arvoja konfliktien havaitsemiseen.

```
public Product UpdateProduct(Product newProduct, Product oldProduct)
{
    using (DemoDataContext dx = new DemoDataContext())
    {
        dx.Products.Attach(newProduct, oldProduct);
        CommitChanges(dx);
    }
    return newProduct;
}
```

Esimerkin Product-entiteettioliolle ei nyt tarvitse määrittää versionumero-ominaisuutta, vaan konfliktien havaitsemiseen voidaan käyttää entiteettiolion mitä tahansa ominaisuuksia.

Esimerkissä 88 on esitelty vastaava asiakasohjelma, jossa välitetään nyt ProductService-luokan UpdateProduct-metodille päivitetty entiteettiolio ja alkuperäinen entiteettiolio.

Esimerkki 88: Asiakasohjelma, jossa entiteettiolion päivittämisessä käytetään alkuperäisiä arvoja.

```
ProductService service = new ProductService();
Product orgProduct = service.GetProductById(1);

Product newProduct = new Product();
newProduct.Id = orgProduct.Id;
newProduct.Name = "BMV Mark III";
newProduct.Price = 250000;
newProduct.CategoryId = orgProduct.CategoryId;

service.UpdateProduct(newProduct, orgProduct);
```

Esimerkistä nähdään, miten asiakasohjelman on säilytettävä alkuperäinen entiteettiolio muistissa ja myös asetettava päivitettävälle entiteettioliolle kaikki ne ominaisuudet, joita käytetään konfliktien havaitsemiseen, eli vaikka asiakasohjelma ei päivitä CategoryId-ominaisuutta, on sen silti asetettava se, koska ominaisuutta käytetään konfliktien havaitsemiseen.

Edellä esitellyt esimerkit käsittelevät irrallisten entiteettiolioiden liittämistä DataContext-luokan muutostenseurantaan hyvin yksinkertaisesti. Entiteettiolioiden liittäminen ja irrottaminen muutostenseurannasta muuttuvat huomattavasti monimutkaisemmaksi, jos sovelluksen pitää mahdollistaa esimerkiksi asiakassovelluksessa tapahtuva entiteettiolion lapsiolioiden päivittäminen, sillä kaikki entiteettioliot pitää liittää muutostenseurantaan aina siten, että myös tarvittavat alkuperäiset arvot välitetään päivitettyjen arvojen yhteydessä. Tällöin joudutaan myös kehittämään mekanismi, jolla voidaan välittää entiteetiolion päivitysmetodille tieto esimerkiksi lapsiolioiden poistosta tai niiden lisäyksestä, jotta päivitysmetodi osaa asettaa lapsioliot oikeaan tilaan tietomallin muutostenseurantaan.

8 Yhteenveto

Suurin osa nykyisistä Enterprise-sovelluksista perustuu pelkästään tiedon hakemiseen tietolähteistä, tiedon esittämiseen käyttäjälle ja tiedon muokkaamiseen sovelluksen liiketoimintalogiikan mukaan. Tällaisissa sovelluksissa tiedonhaku on oleellisessa roolissa ja usein myös suuri osa sovelluksen ohjelmakoodista liittyy pelkästään tiedonhakuun.

Perinteisesti näissä oliopohjaisissa sovelluksissa käytetään tietolähteinä tietokantoja ja XML-tiedostoja, jotka eivät ole oliomuotoisia, jolloin ohjelmoijien on tunnettava myös matalammalla tasolla tietolähteiden toimintaa.

LINQ-hakuarkkitehtuuri kehitettiin juuri nostamaan tiedonhaun abstraktiotasoa ja mahdollistamaan olio-pohjainen lähestymistapa tiedonhakuun, mikä helpottaa oliopohjaisten sovellusten kehittämistä oleellisesti. Abstraktiotason noston asnsiosta ei tarvitse enää miettiä, miten hakea tietoa, vaan voidaan keskittyä enemmänkin tiedon käsittelyyn ja sovelluksen liiketoimintalogiikan toteuttamiseen.

Esimerkiksi sovelluksessa, joka käyttää perinteistä ADO.NET-toteutusta tiedonhakuun tietokannasta, joudutaan pelkästään sovelluksen yksinkertaisen CRUD-toiminnallisuuden toteuttamista varten kirjoittamaan hyvinkin paljon toistuvaa ohjelmakoodia. Hakuarkkitehtuurin tietokantakomponentteja käyttämällä vastaava käsinkirjoitettujen ohjelmakoodirivien ja ohjelmointivirheiden määrä pienentyy murto-osaan, mikä näkyy suoraan sovelluksen kehittämiseen käytettävässä ajassa.

Tietokantaoperaatioiden lisäksi tietokantakomponentit tarjoavat valmiit toteutukset huomattavasti monimutkaisempiin toiminnallisuuksiin, kuten konfliktien hallinnan ja dynaamiset haut, joiden toteuttaminen perinteisin menetelmin on huomattavan työlästä.

Vastaavasti XML-muotoisen tiedon käsittely LINQ:n avulla on perinteisiin tapoihin verrattuna tehokkaampaa. Esimerkiksi hyödyntämällä standardihakuoperaattoreiden tarjoamaa yhteistä rajapintaa tiedonhaussa voidaan vain muutamalla ohjelmakoodirivillä hakea tietokannasta sekvenssi olioita ja yhdistää tähän sekvenssiin XML-tiedostosta haettuja lisätietoja.

Koska hakuarkkitehtuuri mahdollistaa sen käytön laajentamisen uusille tietolähteille, tulee sen käyttö varmasti tulevaisuudessa laajentumaan.

LINQ:n todellinen vaikutus tiedonhakuun tietokeskeisissä sovelluksissa voidaan todeta siitä, että moniin muihinkin ohjelmointikieliin ja ohjelmistokehyksiin on jo toteutettu tai tullaan toteuttamaan vastaavia hakuarkkitehtuureja.

Lähteet

1 Rattz, Joseph C, Jr. Pro LINQ – Language Integrated Query in C# 2008. Apress, 2008.

2 Klein Scott, Professional LINQ. Wrox, 2008.

3 Partial Classes and Methods (C# Programming Guide). (WWW-dokumentti.) MSDN. <http://msdn.microsoft.com/en-us/library/wa80x488.aspx>. Luettu 22.4.2010.

4 Expression Trees: Why LINQ to SQL is Better than NHibernate. (WWW-dokumentti.) Rapid Application Development. <http://rapidapplicationdevelopment.blogspot.com/2008/03/expression-trees-why-linq-to-sql-is.html>. Päivitetty 27.3.2008. Luettu 23.4.2009.

5 Expression Trees. (WWW-dokumentti) MSDN. <http://msdn.microsoft.com/en-us/library/bb397951.aspx> Luettu 23.4.2009

6 Query Expression Syntax for Standard Query Operators. (WWW-dokumentti) MSDN. <http://msdn.microsoft.com/en-us/library/bb882642.aspx>. Luettu 23.4.2009.

7 Classification of Standard Query Operators by Manner of Execution. (WWW-dokumentti) MSDN. <http://msdn.microsoft.com/en-us/library/bb882641.aspx>. Luettu 23.4.2009.

8 Query Expression Basics (C# Programming Guide). (WWW-dokumentti) MSDN. <http://msdn.microsoft.com/en-us/library/bb384065.aspx>. Luettu 28.4.2009.

9 Join Operations. (WWW-dokumentti) MSDN. <http://msdn.microsoft.com/en-us/library/bb397908.aspx>. Luettu 7.5.2009.

10 NHibernate for .NET. (WWW-dokumentti) Hibernate.org. <https://www.hibernate.org/343.html>. Luettu 13.5.2009.

11 The LINQ to SQL Object Model. (WWW-dokumentti). MSDN. <http://msdn.microsoft.com/en-us/library/bb386989.aspx>. Luettu 13.5.2009.

12 Code Generation Tool (SqlMetal.exe). (WWW-dokumentti). MSDN. <http://msdn.microsoft.com/en-us/library/bb386987.aspx>. Luettu 14.5.2009.

13 Object Relational Designer (O/R Designer). (WWW-dokumentti). MSDN. <http://msdn.microsoft.com/en-us/library/bb384429.aspx>. Luettu 14.5.2009.

14 LINQ to Active Directory. (WWW-dokumentti). CodePlex. <http://www.codeplex.com/LINQtoAD>. Luettu 2.7.2009.

15 Responsibilities of the Developer In Overriding Default Behavior (LINQ to SQL). MSDN. <http://msdn.microsoft.com/en-us/library/bb546188.aspx>. Luettu 11.7.2009.

16 Vijay P. Mehta, Pro LINQ Object Relational Mapping with C# 2008. Apress, 2008.

17 Horton Anson, The Evolution Of LINQ And Its Impact On The Design Of C#. MSDN Magazine, kesäkuu 2007. <http://msdn.microsoft.com/fi-fi/magazine/cc163400%28en-us%29.aspx>.

Liite 1: Standardihakuoperaattorien esimerkkien yhteinen lähdekoodi

```
public class ExampleData
{
    private static string[] words =
    {
        "abstract", "event", "new", "struct", "as", "explicit",
        "null", "switch", "base", "extern", "object", "this",
        "bool", "false", "operator", "throw", "break", "finally",
        "out", "true", "byte", "fixed", "override", "try", "case",
        "float", "params", "typeof", "catch", "for", "private",
        "uint", "char", "foreach", "protected", "ulong", "checked",
        "goto", "public", "unchecked", "class", "if", "readonly",
        "unsafe", "const", "implicit", "ref", "ushort", "continue",
        "in", "return", "using", "decimal", "int", "sbyte", "virtual",
        "default", "interface", "sealed", "volatile", "delegate",
        "internal", "short", "void", "do", "is", "sizeof", "while",
        "double", "lock", "stackalloc", "else", "long", "static",
        "enum", "namespace", "string"
    };

    public static int[] Numbers
    {
        get { return Enumerable.Range(0, 100).ToArray(); }
    }

    public static string[] Words
    {
        get { return words; }
    }

    public static List<Product> Products
    {
        get { return CreateProductList(); }
    }

    public static List<Category> Categories
    {
        get { return CreateCategoryList(); }
    }

    private static List<Category> CreateCategoryList()
    {
        return new List<Category>()
        {
            new Category { Id=1, Name="Cars" },
            new Category { Id=2, Name="Books" },
        };
    }

    private static List<Product> CreateProductList()
    {
        return new List<Product>()
        {
            new Product { Id=1, CategoryId=1, Name="BMW",
                Price=100000 },
            new Product { Id=2, CategoryId=1, Name="Ferrari",
                Price=200000 },
            new Product { Id=3, CategoryId=1, Name="McLaren",
                Price=150000 },
            new Product { Id=4, CategoryId=1, Name="Lada" ,
                Price=6500},

            new Product { Id=6, CategoryId=2, Name="Pro Linq",
                Price=23.99M },
            new Product { Id=7, CategoryId=2, Name="Xml", Price=15 },
            new Product { Id=8, CategoryId=2, Name="Linq To Objects",
                Price=19.99M },
            new Product { Id=9, CategoryId=2, Name="Professional C#",
```

```
                Price=9.99M }
        };
    }
}

public class Product
{
    public int Id { get; set; }
    public int CategoryId { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}

public class Category
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Liite 2: Taulukko standardihakuoperaattoreista

Operaattori	Palautusarvon tyyppi	Käyttötarkoitus	Viivästetty	Hakusyntaksi
Aggregate	T	Kooste		
All	bool	Määritys		
Any	bool	Määritys		
AsEnumerable	IEnumerable<T>	Muunnos	Kyllä	
Average	numeerinen arvo	Kooste		
Cast	IEnumerable<T>	Muunnos	Kyllä	from [type] o in
Concat	IEnumerable<T>	Yhdistäminen	Kyllä	
Contains	bool	Määritys		
Count	int	Kooste		
DefaultIfEmpty	IEnumerable<T>	Elementti	Kyllä	
Distinct	IEnumerable<T>	Joukko	Kyllä	
ElementAt	T	Elementti		
ElementAtOrDefault	T	Elementti		
Empty	IEnumerable<T>	Generointi	Kyllä	
Except	IEnumerable<T>	Joukko	Kyllä	
First	T	Elementti		
FirstOrDefault	T	Elementti		

Operaattori	Palautusarvon tyyppi	Käyttötarkoitus	Viivästetty	Hakusyntaksi
GroupBy	IEnumerable <IGrouping<TKey,T>>	Ryhmittely	Kyllä	group [object] by [value] group [object] by [value] into [group]
GroupJoin	IEnumerable<T>	Liitos	Kyllä	join [object] in [from] on [valueA] equals [valueB] into [group]
Intersect	IEnumerable<T>	Joukko	Kyllä	
Join	IEnumerable<T>	Liitos	Kyllä	join [object] in [from] on [valueA] equals [valueB]
Last	T	Elementti		
LastOrDefault	T	Elementti		
LongCount	long	Kooste		
Max	numeerinen arvo tai T	Kooste		
Min	numeerinen arvo tai T	Kooste		
OfType	IEnumerable<T>	Muunnos	Kyllä	

Operaattori	Palautusarvon tyyppi	Käyttötarkoitus	Viivästetty	Hakusyntaksi
OrderBy	IOrderedEnumerable<T>	Järjestäminen	Kyllä	orderby [object]
OrderByDescending	IOrderedEnumerable<T>	Järjestäminen	Kyllä	orderby [object] descending
Range	IEnumerable<T>	Generointi	Kyllä	
Repeat	IEnumerable<T>	Generointi	Kyllä	
Reverse	IEnumerable<T>	Järjestäminen	Kyllä	
Select	IEnumerable<T>	Projektio	Kyllä	select [object]
SelectMany	IEnumerable<T>	Projektio	Kyllä	useita from lauseita
SequenceEqual	bool	Yhdenvertaisuus		
Single	T	Elementti		
SingleOrDefault	T	Elementti		
Skip	IEnumerable<T>	Osittaminen	Kyllä	
SkipWhile	IEnumerable<T>	Osittaminen	Kyllä	
Sum	numeerinen arvo	Kooste		
Take	IEnumerable<T>	Osittaminen	Kyllä	
TakeWhile	IEnumerable<T>	Osittaminen	Kyllä	
ThenBy	IOrderedEnumerable<T>	Järjestäminen	Kyllä	orderby [object1], [object2], ...

Operaattori	Palautusarvon tyyppi	Käyttötarkoitus	Viivästetty	Hakusyntaksi
ThenByDescending	IOrderedEnumerable<T>	Järjestäminen	Kyllä	orderby [obect1], [object2] descending, ...
ToArray	T taulukko	Muunnos		
ToDictionary	Dictionary<T0, T1>	Muunnos		
ToList	IList<T>	Muunnos		
ToLookup	ILookup<T0, T1>	Muunnos		
Union	IEnumerable<T>	Joukko	Kyllä	
Where	IEnumerable<T>	Rajoitus	Kyllä	where [expression]