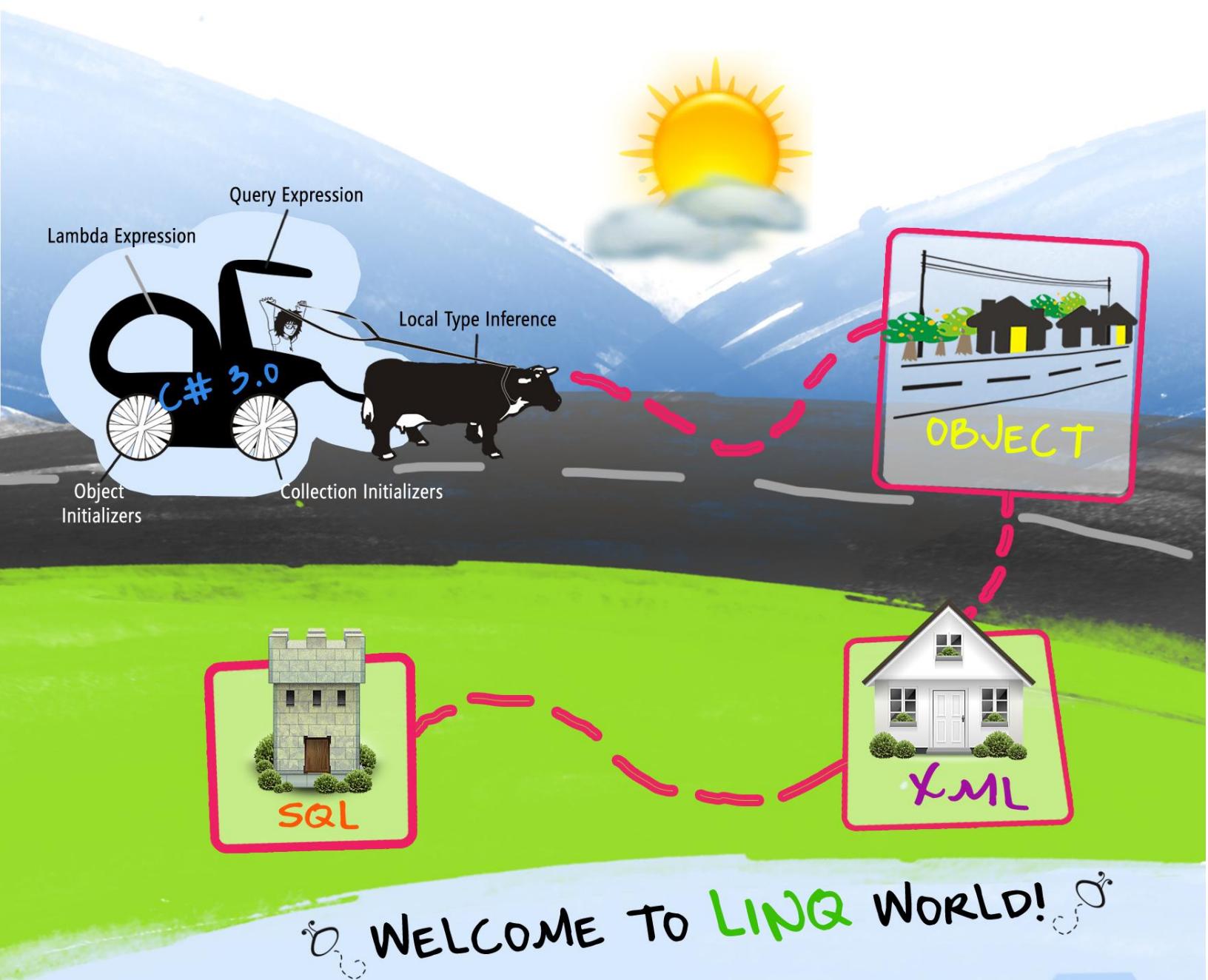


Perjalanan ke desa LINQ



Apa itu LINQ?

Fitur baru dalam C# 3.0

Mengenal LINQ to Object, LINQ to XML, dan LINQ to SQL



Kata Pengantar

Puji syukur penulis panjatkan kepada Tuhan Yang Maha Esa, karena atas tuntunan dan kasih-Nya penulis mampu menyelesaikan ebook ini. Buku dengan judul perjalanan ke desa LINQ ini mengusung tema perjalanan yang akan membawa pembaca ke sebuah desa dengan nama desa LINQ. Dalam buku ini saya banyak menggunakan bahasa yang santai agar para pembaca tidak merasa sedang membaca sebuah text book melainkan sebuah cerita.

Dalam buku ini pun penulis menggunakan Visual C# Express 2008 sebagai editor kode dan Microsoft SQL Server 2005 sebagai basis data, sehingga selain ebook ini gratis semua tools yang digunakan pun gratis. Diharapkan dengan tersedianya material pembelajaran dan tools yang gratis para developer indonesia dapat semakin meningkatkan skill nya dan memajukan kualitas IT Indonesia.

Ucapan terima kasih yang sebesar-besarnya saya ucapkan kepada Tim DPE microsoft Indonesia dan bang Narendra Wicaksono pada khususnya karena atas tuntunan dan tututan beliau lah saya dapat menyelesaikan ebook ini. Kemudian kepada seluruh Kru MIC – ITB dan pada khususnya Ume yang telah menjadi desainer dari seluruh icon yang digunakan pada ebook ini. Terima kasih pun tidak lupa saya ucapkan pada kawan-kawan MSP se-Indonesia khususnya Fajar rekan MSP saya di ITB yang juga telah merilis buku Windows Server 2008.

Ada pepatah bahwa tidak ada gading yang tidak retak maka buku ini pun tidak lepas dari kesalahan sehingga jika ada saran dan kritik yang membangun anda dapat mengirimkan email kepada saya di ronald@mic.itb.ac.id. Akhir kata penulis ingin mengucapkan terima kasih kepada para pembaca yang telah mengunduh ebook ini kemudian menyempatkan waktunya untuk membacanya dan Selamat menikmati perjalanan di desa LINQ.

Februari 2008

Ronald Rajagukguk

Daftar Isi

Kata Pengantar	2
Panduan Sebelum Memulai Perjalanan	5
Kebutuhan perangkat keras :	5
Kebutuhan perangkat lunak.....	5
LINQ Overview	6
Fitur baru dalam C# 3.0.....	9
Lambda Expression	9
Kata kunci var, inisialisasi objek dan koleksi dan tipe anonymous	10
Extension methods	12
Partial methods.....	15
Query Expressions.....	15
Penutup	16
LINQ To Object.....	17
Sintaks Linq	18
Sintaks LINQ secara komplit.....	20
Standard Query Operators.....	21
Projection.....	21
Restriction.....	24
Ordering.....	25
Grouping	28
Aggregate.....	32
Penutup	43
LINQ to XML.....	44
LINQ to XML API.....	50
Pembuatan XML.....	51
Penulisan XML.....	55
Pembacaan XML	59
Penelusuran Dokumen XML.....	63

Modifikasi XML	73
Penutup	90
LINQ to SQL.....	91
Dasar LINQ to SQL.....	93
Pembuatan kelas entitas.....	94
Pembuatan DataContext.....	95
Penggunaan Tools pendukung LINQ to SQL	96
Visual Studio Object Relational Designer	96
SQL Metal.....	100
Operasi terhadap basis data dengan LINQ to SQL.....	103
Pemasukan data dengan menggunakan LINQ to SQL	103
Melakukan pengubahan data (Update) dengan menggunakan LINQ to SQL	107
Penghapusan Data (delete)dengan menggunakan LINQ to SQL	110
Isu Konkurensi.....	112
Penutup	117
LAMPIRAN A	118
LAMPIRAN B.....	126

Panduan Sebelum Memulai Perjalanan

Sebelum memulai perjalanan ini maka ada baiknya anda mempersiapkan benda-benda dibawah ini. sehingga perjalanan kita akan berjalan secara lancar.

Kebutuhan perangkat keras :

Perangkat keras yang anda butuhkan agar eksplorasi kita dalam buku ini berjalan lancar adalah :

1. Komputer dengan kecepatan 1GHz atau lebih.
2. RAM 512 atau lebih.
3. Ruang kosong di harddisk sebesar minimal 2GB
4. Monitor dengan minimal tampilan layar 1024x768
5. Keyboard, mouse dan pelengkap lainnya.

Kebutuhan perangkat lunak

Perangkat lunak yang anda butuhkan dalam eksplorasi ini adalah :

1. Salah satu dari Sistem Operasi berikut
 - o Windows XP Professional SP2 x86/x64 (WOW).
 - o Windows 2003 SP1 dengan berbagai variannya (R2, x86, x64 (WOW)).
 - o Windows Vista Business, Home Premium, Ultimate atau Enterprise.
 - o Windows 2008
2. Tools Pengembangan perangkat lunak
 - a. Microsoft Visual C# 2008 Express Edition
 - b. Microsoft SQL Server Express 2005
 - c. SQL Server Management Studio Express 2005

Sebagai informasi anda dapat memperoleh ketiga tools tersebut secara GRATIS, yak sekali lagi GRATIS di situs microsoft(<http://www.microsoft.com/express/>).

Baiklah jika anda telah mempersiapkan semua syarat yang dituliskan diatas maka mari sekarang kita mulai perjalanan kita mengarungi desa LINQ.

LINQ Overview

Pada awal perjalanan ini penulis ingin mengikuti tradisi yang sudah mendarah daging dalam setiap buku yang mempelajari dunia programming. Di awal pembelajaran tentunya dimulai dengan bagaimana menuliskan “hello world”. Maka karena penulis tidak ingin melanggar tradisi silahkan ketikkan kode dibawah ini untuk menampilkan “hello world” ala LINQ.

```
class Program
{
    static void Main(string[] args)
    {
        string[] helloWord={ "hello", "this", "is",
                            "LINQ", "by", "Ronald", "Rajagukguk" };
        var results = from word in helloWord
                      where(word[0]>'g' || (word[0]>'K' && word[0]<'M'))
                      select word+ ' ';
        foreach(var wordItem in results)
            Console.WriteLine(wordItem);
    }
}
```

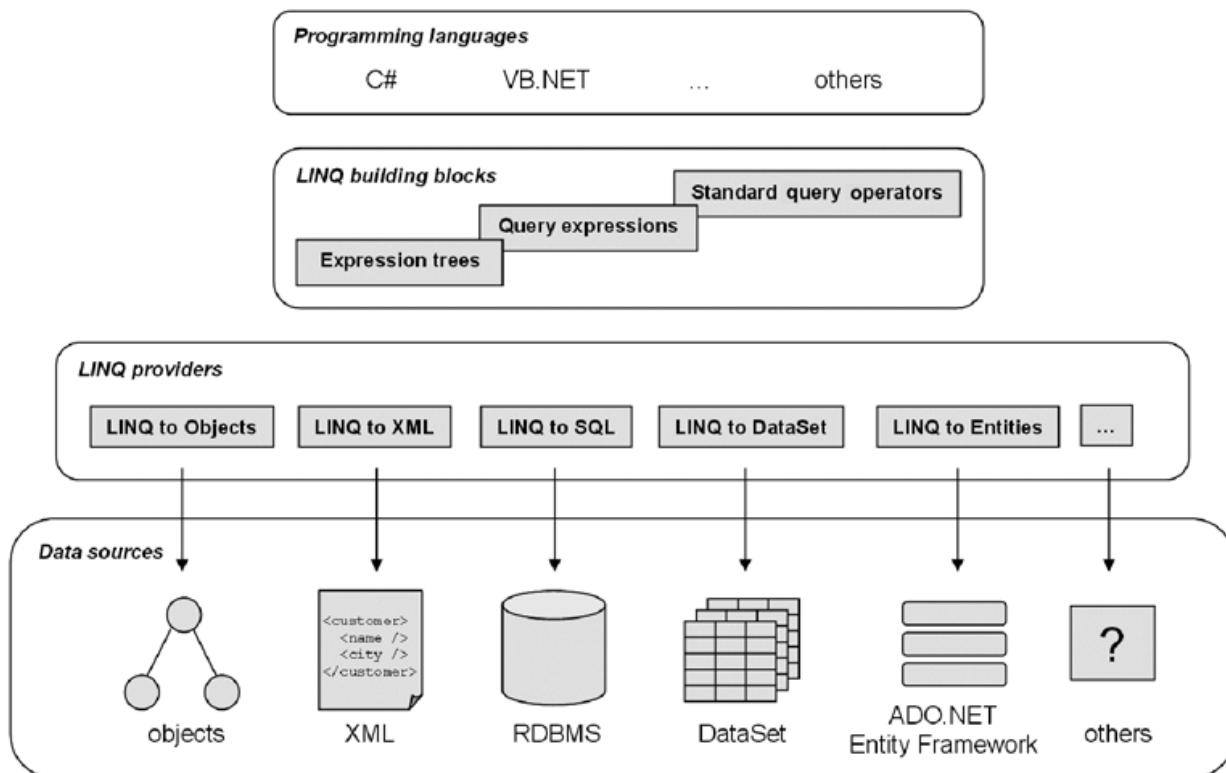
Kode aneh bin ajaib diatas akan menuliskan “hello this is LINQ” pada layar konsole anda. Kode diatas telah menggunakan apa yang disebut LINQ. LINQ adalah singkatan dari Language Integrated Query, teknologi ini pertama kali dikenalkan oleh Anders Hejlsberg dalam Microsoft Professional Developers Conference (PDC) 2005. Secara singkat LINQ dapat didefinisikan sebagai sebuah metode yang memudahkan dan menseragamkan cara pengaksesan data. Sehingga dengan menggunakan LINQ developer hanya perlu menggunakan sebuah teknik pengaksesan data saja. Karena jika kita lihat dunia pemrogramman saat ini maka umumnya developer perlu menguasai skill yang berbeda-beda untuk setiap media penyimpanan data.

Sebagai contoh ketika sedang berurusan dengan basis data tentunya developer perlu memiliki kemampuan SQL yang mumpuni dan di lain waktu ketika harus berurusan dengan XML developer pun harus memiliki pengetahuan yang cukup tentang cara pengaksesan XML yang tentunya sangat berbeda dengan basis data. Disini lah letak keunggulan LINQ, LINQ hadir untuk menseragamkan cara pengaksesan data tersebut. Sehingga dengan menggunakan LINQ kita hanya cukup menguasai satu saja teknik pengaksesan data untuk berbagai media penyimpanan data (XML, Basis data, dan lainnya). Mengutip sebuah iklan minuman di televisi maka dapat dibuat slogan “Apapun datanya, ngaksesnya tetap pake LINQ”.

Baik, mari sekarang kita kembali sejenak ke kode hello world ala LINQ yang ada pada awal bab. Jika anda seorang developer yang telah berpengalaman dengan C# 2.0 tentunya akan merasa janggal dengan kode diatas. Ada 3 kata kunci baru yang tertulis pada kode diatas yang umum kita temui pada Structured Query Language (SQL) yaitu from, where dan juga select. Kemudian ada satu lagi yaitu var, kata ini tentunya umum ditemui pada javascript dan beberapa bahasa lainnya. Apakah yang gerangan terjadi di

kode diatas? Apakah SQL telah masuk menjadi C#? jawabannya adalah hal tersebut terjadi karena adanya integrasi LINQ ke dalam bahasa pemrogramman yang dalam kasus ini C#. C# diedisi yang paling baru (C# 3.0) telah menambahkan beberapa fitur dalam rangka mendukung LINQ.

Kemudian selain integrasi yang kuat terhadap bahasa pemrogramman LINQ pun menambahkan sekumpulan API yang memungkinkan pengaksesan data terhadap sumber data yang berbeda-beda. Arsitektur LINQ secara global dapat dilihat pada gambar dibawah ini.



Dalam gambar diatas terlihat jelas bahwa pada bagian paling atas LINQ mengintegrasikan dirinya dengan bahasa pemrogramman yang sekarang ini baru ada dua buah yaitu C# 3.0 dan Visual Basic 9.0. Kemudian pada bagian bawahnya ada bagian provider. Bagian ini adalah bagian yang mengizinkan LINQ untuk melakukan pengaksesan terhadap format data yang berbeda-beda. LINQ mengizinkan kita untuk membuat sebuah provider lainnya, sehingga misalnya kita ingin membuat sebuah provider yang menjembatani LINQ untuk melakukan pengaksesan terhadap file sistem hal tersebut dapat kita lakukan. Namun buku ini secara khusus akan membahas 3 buah API dari provider tersebut yaitu :

1. LINQ to Object

Merupakan sekumpulan API yang mengandung sejumlah *standard query operators (SQO)* untuk mengambil data dari sembarang objek yang mengimplementasikan interface `IEnumerable<T>`. *Query* ini digunakan terhadap data yang ada dalam memori.

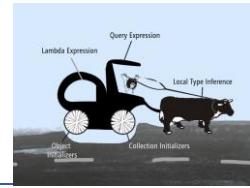
2. LINQ to ADO .NET

Merupakan API yang memberikan kemampuan SQO untuk dapat bekerja terhadap basis data Relational. Ada 3 sub bagian dalam LINQ to ADO .NET yaitu LINQ to SQL untuk melakukan *query* terhadap database relasional seperti Microsoft SQL Server, LINQ to Dataset untuk melakukan *query* terhadap data sets dan data tables yang ada pada ADO .NET dan terakhir LINQ to Entities yang merupakan solusi dari Microsoft ORM yang memungkinkan pengembang perangkat lunak menggunakan Entities secara deklaratif dalam menspesifikasi struktur dari objek bisnis dan menggunakan LINQ untuk melakukan *query* terhadapnya.

3. LINQ to XML

Merupakan API yang memberikan kemampuan SQO (Standard Query Operator) untuk bekerja terhadap XML termasuk didalamnya kemampuan untuk membuat sebuah XML document.

Merasa tertarik dengan LINQ dan segala fitur yang ditawarkannya? Mari kita mulai perjalanan kita mengarungi dunia LINQ yang sangat menarik dan akan membawa kita ke sebuah dunia pengaksesan data yang baru ini.



Fitur baru dalam C# 3.0

Sebelum memulai perjalanan mengarungi dunia LINQ izinkan saya untuk memperkenalkan C# 3.0 . C# 3.0 adalah kendaraan yang akan kita gunakan selama mengarungi desa LINQ. Seperti telah dibahas pada bagian overview C# merupakan salah satu bahasa yang telah mendukung LINQ selain VB 9.0. Bagaimana C# dapat mendukung LINQ maka kita perlu melihatnya terlebih dahulu penambahan fitur dalam bahasa ini.

Secara total jika melihat MSDN for Visual Studio 2008 ada 9 fitur baru yang ditambahkan dalam C# 3.0. Namun dalam inspeksi kendaraan kali ini saya secara khusus akan membaginya menjadi 5 poin yaitu :

1. Lambda Expression
2. Kata kunci var, inisialisasi objek dan koleksi dan tipe anonymous
3. Extension methods
4. Partial methods
5. Query Expressions

Baiklah mari sama-sama kita mulai inspeksi kita terhadap C# 3.0 ini.

Lambda Expression

Jika anda sama seperti saya yang pernah mempelajari pemrogramman fungsional dengan menggunakan LISP, pasti anda akan akrab dengan istilah Lambda. Pada dasarnya Lambda Expression adalah sebuah anonymous function yang mengandung sejumlah ekspresi dan pernyataan yang dapat digunakan untuk membuat sebuah tipe delegate ataupun sebuah pohon ekspresi. Sehingga kode C# 2.0 dibawah ini :

```
delegate int tambahSatu(int p);
int penambah(int p)
{
    return ++p;
}
static void Main(string[] args)
{
    tambahSatu ts = new Program().penambah;
    Console.WriteLine(ts(12));
}
```

Dapat digantikan pada C# 3.0 yang telah menggunakan Lambda expresion menjadi sebagai berikut:

```
delegate int tambahSatu(int p);
static void Main(string[] args)
{
    tambahSatu ts = (x => ++x);
    Console.WriteLine(ts(12));
}
```

Terlihat dalam kode C# 3.0 kode (`x=>+x`) telah menggantikan pendefinisian sebuah fungsi penambah yang ada dalam kode C# 2.0, disinilah letak dari pesan lambda expression. Terlihat pada kode diatas ada sebuah operator baru yaitu (`=>`) operator tersebut merupakan lambda operator. Sebagai informasi bagian kiri dari operator lambda adalah parameter masukan sedangkan bagian kanan adalah ekspresi ataupun pernyataan. Sehingga pada contoh kode C# 3.0 diatas variabel `x` yang ada disebelah kiri lambda operator merupakan masukan parameter masukan yang berupa integer, kemudian pada bagian kanan dituliskan sebuah expresi yang menambah integer masukan tersebut dengan nilai 1. Maka ketika tipe delegate tambahSatu dieksekusi maka akan dikembalikan sebuah nilai integer yaitu 13.

Kata kunci var, inisialisasi objek dan koleksi dan tipe anonymous

Sekarang kita akan menginspeksi 3 hal baru dalam C# 3.0. Mengapa saya memasukkan 3 hal ini dalam 1 inspeksi? Alasannya adalah karena sebenarnya 3 hal baru sangat-sangat berkaitan erat satu sama lainnya ketika sedang membicarakan LINQ. Sebelum kita mulai membicarakan setiap fitur ini secara lebih detail ada baiknya kita lihat kode di bawah ini :

```
var1 pegawai = new {2 NamaDepan = "ronald"3, NamaBelakang = "rajagukguk"3 };
```

kode diatas telah menggunakan apa yang disebut ¹kata kunci var yang akan secara otomatis mendeteksi tipe dari objek yang ditampungnya. ²pendeklarasian sebuah tipe anonymous dan ³inisialisasi objek tanpa memanggil konstruktor dari objek tersebut. Baiklah mari sekarang kita inspeksi hal tersebut secara lebih mendetail :

Local Type Inference

Untuk mendukung fitur ini maka pada C# 3.0 dikenalkan sebuah kata kunci baru yaitu var. Sehingga jika sebelumnya ketika anda melakukan pendeklarasian variabel anda harus menuliskan terlebih dahulu tipe dari variabel tersebut, maka sekarang anda dapat menggunakan kata kunci var untuk menggantikannya dan biarkan kompiler yang akan melakukan inferensi dari tipe nilai variabel tersebut untuk mengganti kata var dengan tipe sebenarnya dari variabel anda. Contohnya adalah pada kode dibawah ini :

```
var nilai = "ronald";
```

Pada kode diatas kompiler C# akan secara otomatis menentukan tipe dari variabel nilai berdasarkan tipe nilai yang ada di sebelah kanan yaitu string, sehingga jika anda menggunakan Visual Studio 2008 sebagai IDE, jika anda *hover* di atas tulisan `var` akan muncul *popup* sebagai berikut yang menandakan bahwa kompiler telah melakukan inferensi dan menganggap bahwa tipe dari variabel nilai adalah string.

```
var nilai = "ronald";
class System.String
Represents text as a series of Unicode characters.
```

Namun sebagai catatan karena sifat C# yang masih strongly typed kita harus menginisialisasi nilai variabel tersebut langsung pada saat pendeklarasian variabel. Sehingga tentunya kode dibawah ini akan menimbulkan pesan kesalahan ketika dilakukan kompilasi :

```
var pegawai;
    Implicitly-typed local variables must be initialized
```

Kemudian tidak seperti penggunaan kata kunci var pada bahasa script seperti java script kita tidak dapat mengganti tipe dari variabel yang didefinisikan dengan kata kunci var, sehingga kode dibawah ini pun akan memimbulkan pesan kesalahan.

```
var nama ="ronald";
nama = 12;
    Cannot implicitly convert type 'int' to 'string'
```

Object Initializers

Fitur ini mengizinkan anda untuk mengisi nilai dari field atau properti dari sebuah objek pada saat pembuatannya tanpa perlu secara eksplisit memanggil konstruktor. Contoh penggunaanya adalah pada kode berikut :

```
private class Binatang
{
    public int jumlahKaki { get; set; }
    public string Nama { get; set; }
}

static void Main(string[] args)
{
    Binatang kuda = new Binatang { jumlahKaki = 3, Nama = "Kuda" };
}
```

Pada kode diatas terlihat jelas bahwa pada saat inisialisasi objek Binatang di dalam kurung kurawal dilakukan pengisian terhadap field jumlahKaki dan Nama. Pada versi sebelumnya untuk melakukan hal ini anda tentunya harus membuat sebuah konstruktor khusus yang menerima masukan dua parameter yang kemudian mengisikan nilai di parameter tersebut ke field atau properti yang sesuai. Namun seperti telah terlihat dalam kode diatas sekarang hal tersebut dapat dilakukan secara ringkas dengan menggunakan fitur Object Initializers.

Collection Initializers

Fitur ini mengizinkan anda untuk dapat mengisi sebuah koleksi (kelas yang mengimplementasikan I Enumerable) pada saat inisialisasi koleksi tersebut. Sehingga efeknya anda tidak perlu lagi memanggil prosedur Add untuk menambahkan sebuah objek kepada koleksi tersebut. Contoh penggunaannya adalah pada kode dibawah ini.

```
List<string> koleksi = new List<string>{"satu", "dua", "tiga"};
```

Kode diatas dimaksudkan untuk menambahkan 3 buah objek string kedalam objek List yang menjadi penampungnya. Terlihat bahwa sekarang anda tidak perlu lagi memanggil prosedur **add** untuk menambahkan isi dari koleksi. Anda dapat langsung memasukkan objek yang ingin anda tambahkan pada saat inisialisasi objek yang menjadi penampung data.

Anonymous Types

Dengan menggunakan Anonymous Types maka sekarang anda dapat membuat sebuah struct tanpa nama pada saat *runtime*. Namun karena struct ini tidak memiliki nama pada kode anda harus menampung struct ini dengan menggunakan kata kunci var. sebagaimana contoh kode dibawah ini :

```
var tanpaNama = new { Nama = "Ronald", Umur = 22 };
```

Kode diatas akan membuat sebuah struct yang memiliki dua buah properti yaitu Nama dan Umur dengan tipe string dan integer. Sehingga jika anda menggunakan Visual Studio 2008 dan melakukan *hover* diatas kata var akan muncul *popup* yang mengandung informasi bahwa variabel tanpa nama merupakan sebuah Anonymous Types dengan dua buah properti.

```
var tanpaNama = new { Nama = "Ronald", Umur = 22 };
AnonymousType 'a
Anonymous Types:
'a is new { string Nama, int Umur }
```

Extension methods

Proses inspeksi kita sampai ke sebuah fitur yang cukup kontroversial jika anda adalah seorang pemuja OOP (Object Oriented Programming) yang cukup ekstrim. Karena adanya konsep ini tentunya akan melanggar konsep OOP jika kita menganggap bahwa C# adalah bahasa yang **hanya** mendukung OOP. Sekontroversial apakah fitur ini? sebelumnya mari segarkan otak kita dengan konsep OOP yang cukup terkenal yaitu inheritance (pewarisan sifat), umumnya ketika developer ingin menambahkan sebuah fungsi pada kelas namun tidak ingin merubah kelas tersebut ia akan menggunakan inheritance untuk membuat sebuah kelas turunan. Kemudian menambahkan fitur baru tersebut di kelas turunan yang dibuatnya tadi. Contohnya seperti dibawah ini :

```
delegate int tambahSatu(int p);
int penambah(int p)
{
    return ++p;
}
static void Main(string[] args)
{
    tambahSatu ts = new Program().penambah;
    Console.WriteLine(ts(12));
}
```

Pada kode diatas kita akan menambahkan sebuah fitur baru yaitu terbang, namun karena kita tidak ingin merubah kelas Unggas (mengingat tidak semua unggas bisa terbang) maka kita menurunkan kelas unggas dan membuat kelas Burung dan menambahkan fitur tersebut pada kelas burung. Hal ini tentunya umum di kalangan pengikut OOP. Namun mari kita lihat fitur kontroversial ini in action saat akan melakukan hal seperti diatas :

```
class Unggas
{
    public void jalan()
    {
        Console.WriteLine("jalan");
    }
}
static class Penambah
{
    public static void terbang(this Unggas kelas)
    {
        Console.WriteLine("terbang");
    }
}
class Program
{
    static void Main(string[] args)
    {
        Unggas burung = new Unggas();
        burung.terbang();
    }
}
```

Satu kata WOW! Terlihat diatas bahwa sekarang kelas Unggas dapat memiliki fungsi terbang, meskipun kita tidak mendeklarasikan fungsi tersebut dalam definisi kelas. Kita mendefinisikan fungsi tersebut pada kelas static bernama Penambah dan disinilah letak Extension Method beraksi.

Seperti terlihat pada kode diatas dengan Extension Methods maka method yang baru tersebut akan seolah-olah ditempelkan ke sebuah kelas, cara penggunaanya ialah dengan membuat sebuah kelas static yang mengandung fungsi yang akan ditambahkan. Namun method baru yang akan ditambahkan tersebut harus bersifat static dan pada parameter pertamanya memiliki masukan berupa objek dari kelas yang akan ditambahkan method ini selain itu diawal parameter pertama perlu ditambahkan kata kunci this. Kemudian parameter selanjutnya akan berlaku sebagai parameter dari method yang baru ini. Contohnya pada kode dibawah ini :

```

static class Penambah
{
    public static void terbangJauh(this Unggas kelas, int jarak)
    {
        Console.WriteLine("terbang {0} kilometer", jarak);
    }
}
class Program
{
    static void Main(string[] args)
    {
        Unggas burung = new Unggas();
        burung.terbangJauh(3);
    }
}

```

Akan menghasilkan method terbangJauh dengan satu parameter pada kelas Unggas.

Extension methods dapat menempelkan method baru pada semua objek, sehingga dengan keadaan seperti ini maka kita pun dapat memperluas kelas yang ada pada framework .NET, sebagai mana dicontohkan pada kode dibawah ini :

```

static class Penambah
{
    public static string TigaHurufPertama(this string kata)
    {
        string result;
        if (kata.Length < 3)
            result = kata;
        else
            result = kata.Substring(0, 3);
        return result;
    }
}
class Program
{
    static void Main(string[] args)
    {
        string nama = "ronald";
        Console.WriteLine(nama.TigaHurufPertama());
    }
}

```

Menarik bukan? Dalam contoh diatas kita menambahkan fungsi TigaHurufPertama pada sebuah kelas string! Fitur ini bahkan tidak memerlukan kode dari kelas yang akan ditambahkannya (kita tentunya tidak memiliki source code dari kelas string pada .NET framework). Namun ada satu hal yang perlu diingat pada extension method ialah bahwa kelas static tempat pendeklarasian extension method ini harus berada dalam cakupan yang sama dengan kode yang menggunakannya, atau dalam artian berada dalam satu namespace dengan kode yang menggunakannya. Jika tidak berada dalam satu namespace

tentunya kita dapat menggunakan kata kunci using untuk mengimpor namespace yang menjadi tempat pendefinisian dari extension method tersebut.

Banyak developer yang bertanya, dimanakah letak Extension Methods ini pada LINQ. Jawabannya adalah tim pengembang LINQ menggunakan extension methods untuk menambahkan Standar Query Operator pada interface `IEnumerable<T>`, sehingga kita dapat menggunakan SQO LINQ pada setiap kelas yang mengimplementasikan interface `IEnumerable`.

Partial methods

Dalam C# 2.0 dikenal sebuah konsep partial class dengan kata kunci partial yang memungkinkan pembagian definisi kelas pada lebih dari satu file. Ambil contoh misalnya anda memiliki sebuah kelas Binatang maka pada file A.cs anda dapat menuliskan kelas binatang yang mengandung fungsi jalan dan pada file B.cs anda dapat menuliskan kelas binatang yang mengandung fungsi duduk. Kemudian jika anda melakukan kompilasi dengan dua buah file tersebut maka anda akan memperoleh kelas Binatang yang memiliki fungsi jalan dan duduk. Konsep yang cukup menarik ini dilanjutkan dengan konsep Partial Methods pada C# 3.0. Dengan menggunakan partial method maka anda dapat menuliskan sebuah definisi dari method di sebuah kelas partial dan kemudian menuliskan implementasi dari method tersebut di kelas partial yang lain. Contohnya adalah sebagai berikut :

```
//ditulis pada file1.cs
partial void Jalan();
//ditulis pada file2.cs
partial void Jalan()
{
    Console.WriteLine("Jalan");
}
```

Dalam contoh diatas terlihat bahwa pada file1.cs dituliskan definisi dari method Jalan() untuk kemudian pada file2.cs dituliskan implementasi dari method tersebut. Untuk anda para C/C++ programmer tentunya hal ini mengingatkan pada pendefinisian method dari kelas yang umum ditulis pada file .H dan implementasinya yang ditulis pada file .cpp atau .c

Query Expressions

Query Expressions merupakan sebuah fitur yang memungkinkan C# untuk dapat menuliskan sintaks mirip seperti SQL pada contoh di awal buku ini. Sebenarnya dengan menggunakan Query Expression maka kode dibawah ini :

```
var results = from word in helloWord
              where (word[0] > 'g' ||
                     (word[0]>'K' && word[0]<'M' ))
```

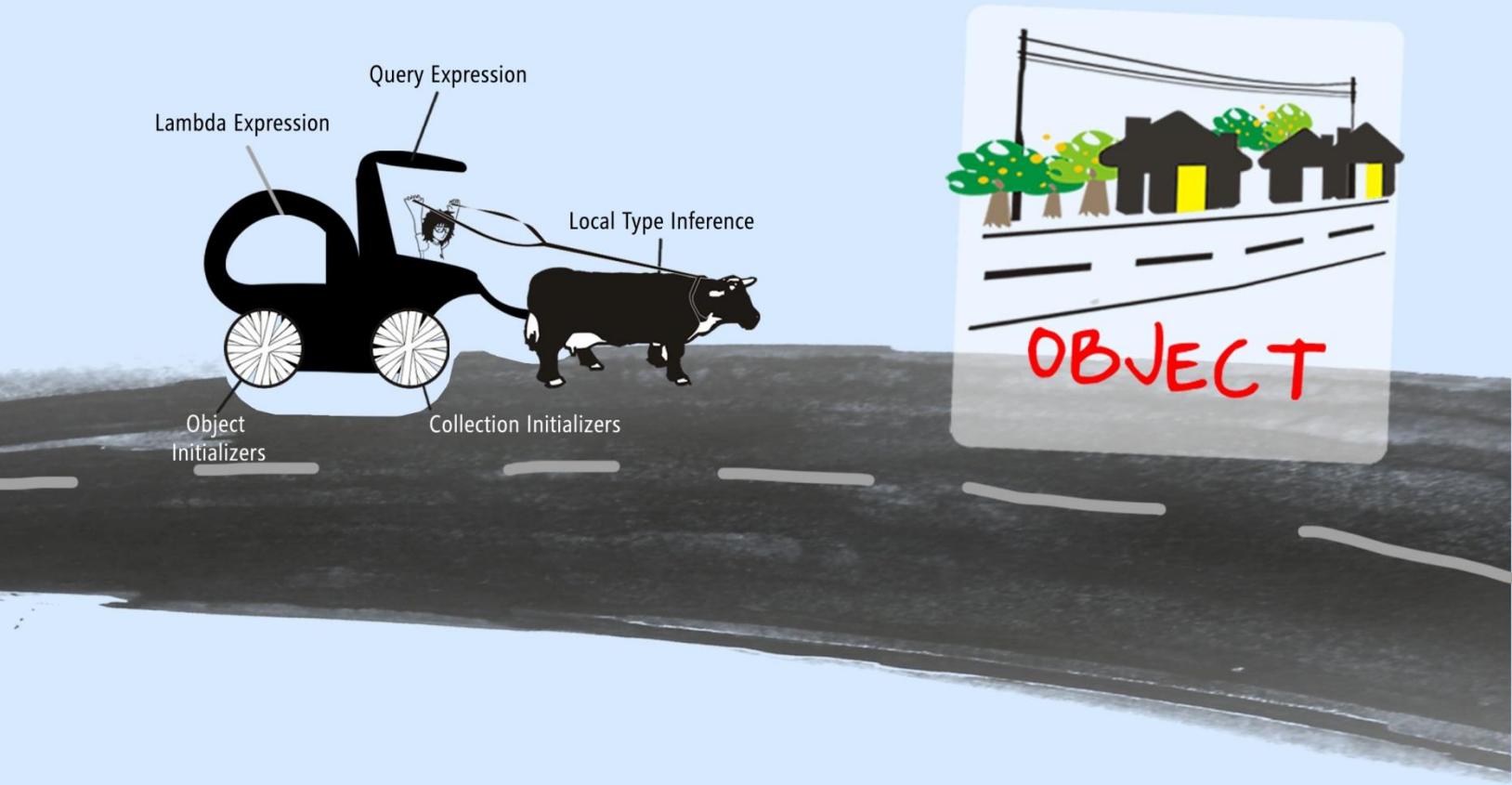
Akan diubah menjadi

```
var results = helloWord.Where(
    word => (
        (word[0] > 'g' ||
        (word[0] > 'K' && word[0] < 'M'))))
    .Select(word => word + ' ');
```

Terlihat bahwa pada kode diatas bahwa setiap klausanya dalam query kita akan diterjemahkan menjadi sebuah method yang menerima masukan berupa lambda expression. Sehingga pada akhirnya seluruh query kita akan diterjemahkan menjadi serentetan pemanggilan fungsi SQO. Tentunya adanya Query Expression akan memudahkan kita ketika menuliskan query dengan LINQ.

Penutup

C# pada edisinya yang ketiga ini telah memberikan beberapa fitur menarik yang memungkinkan LINQ untuk diterapkan. Selain digunakan untuk LINQ anda pun dapat menggunakan fitur-fitur tersebut untuk keperluan lainnya. Sebagai tambahan karena C# adalah bahasa maka jika anda menggunakan Visual Studio 2008 dan memilih .NET Framework 2.0 fitur dalam C# 3.0 ini masih akan tetap dapat digunakan dengan baik. Namun perlu dicatat meskipun C# 3.0 dapat berjalan di .NET Framework 2.0 tapi API untuk LINQ tidak akan bisa dipakai karena API untuk LINQ hanya ada mulai .NET 3.5.



LINQ to Object

LINQ To Object

Perjalanan pertama kita akan dimulai dengan mengunjungi pondok LINQ to Object. Salah satu alasan yang membuat LINQ menjadi teknologi yang sangat menarik adalah integrasinya yang sangat kuat terhadap sintaks bahasa yang dalam hal ini C# 3.0 atau VB 9.0. Teknologi LINQ didasari oleh sekumpulan operator query atau yang sering disebut Standard Query Operators (SQO). SQO didefinisikan sebagai sekumpulan extension methods yang akan bekerja kepada setiap objek yang mengimplementasikan `IEnumerable<T>`. Sehingga dengan pedekatan seperti ini maka kita dapat menggunakan LINQ untuk melakukan query terhadap objek yang ada dalam framework, sehingga dengan kata lain kita tidak perlu merubah kode-kode kita hanya sekedar agar LINQ bisa dioperasikan terhadap kumpulan objek kita, objek disini dapat didefinisikan sebagai objek yang ada di memory.

Sebagai contoh jika anda memiliki sebuah array yang berisikan integer dan memerlukan pengurutan nilai tersebut mulai dari yang terkecil sampai yang terbesar dan ingin menggunakan sintaks yang mirip dengan Query SQL dalam melakukan hal ini, atau anda telah memiliki sebuah `ArrayList` yang berisikan objek `Pelanggan` dan ingin menemukan sebuah pelanggan dengan kriteria tertentu dan ingin menggunakan Query SQL dalam melakukan hal ini. Maka LINQ to Object adalah yang anda butuhkan!

Sintaks Linq

Sebelum memulai perjalanan eksplorasi LINQ to Object Untuk lebih memahami cara penggunaan LINQ maka ada baiknya kita mendalami dulu sintaks dari LINQ ini. Sebagai contoh mari kita mulai dengan sedikit contoh. Pertama-tama buat sebuah kelas Mahasiswa sebagai berikut :

```
public class Mahasiswa
{
    public string Nama{ get; set; }
    public int NIM{ get; set; }
    public double IPK{ get; set; }
}
```

Sekarang anggap bahwa anda ingin melakukan query terhadap sekumpulan objek mahasiswa untuk mencari mahasiswa yang memiliki IPK lebih besar dari 3, maka dengan menggunakan LINQ anda dapat melakukannya sebagaimana dicontohkan pada kode dibawah ini :

```
using System;
using System.Collections.Generic;
using System.Linq;

class Program
{

    static void Main(string[] args)
    {
        Mahasiswa[] kumpulanMahasiswa = new Mahasiswa[]
        {
            new Mahasiswa() {Nama="Joko",NIM="13504117",IPK=3.8},
            new Mahasiswa() {Nama="Saseno",NIM="13504002",IPK=3.0},
            new Mahasiswa() {Nama="Udin",NIM="13504009",IPK=3.2}
        };
    }
}
```

```

        IEnumerable<string> hasil = from m in kumpulanMahasiswa
                                      where m.IPK > 3.1
                                      select m.Nama;
        foreach(string mahasiswa in hasil)
            Console.WriteLine(mahasiswa);

    }
}

```

Hasil dari kode diatas adalah :

```
C:\Windows\system32\cmd.exe
Joko
Udin
Press any key to continue . . .
```

Sintaks dari query LINQ seperti pada kode diatas (kode yang dihighlight kuning) mirip sekali dengan statement SQL yang biasa dipakai dalam basis data bukan? meskipun gaya penulisannya sedikit berbeda. Untuk lebih memahami tentang sintaks LINQ maka mari sama-sama kita coba mendekontruksi sintaks ini.

Pertama-tama kita akan memilih sebuah objek dari sebuah objek dari kelas yang mengimplementasikan `IEnumerable<T>` melalui sintaks ini :

```
from m in kumpulanMahasiswa
```

Kemudian kita akan melakukan filter terhadap objek tersebut apakah memenuhi kriteria kita atau tidak (Kriteria yang kita inginkan adalah objek Mahasiswa tersebut memiliki IPK lebih dari 3,1) melalui sintaks ini :

```
where m.IPK > 3.1
```

Lalu setelah itu kita memutuskan untuk hanya mengambil properti Nama dari objek mahasiswa tersebut melalui sintaks ini :

```
select m.Nama
```

Kita pun sebenarnya dapat menuliskan ekspresi query diatas menjadi seperti dibawah ini :

```
 IEnumerable<string> hasil = kumpulanMahasiswa.
                                Where(x => x.IPK > 3.1).
                                Select(x => x.Nama);
```

Kedua metode Where dan Select akan menerima masukan sebuah Lambda Expression. Lambda expresion ini diterjemahkan menjadi sebuah tipe delegate seperti yang ditetapkan dalam namespace `System.Linq` seperti dibawah ini :

```
public delegate T Func<T>();
public delegate T Func<A0, T>(A0 arg0);
```

```
public delegate T Func<A0, A1, T>(A0 arg0, A1 arg1);
public delegate T Func<A0, A1, A2, T>(A0 arg0, A1 arg1, A2 arg2);
public delegate T Func<A0, A1, A3, T>(A0 arg0, A1 arg1, A2 arg2, A3
arg3);
```

Sebagai informasi pada SQO ada banyak sekali prosedur yang menerima masukan tipe delegate diatas sebagai parameternya. Query awal kita pun jika ditulis dengan memanfaatkan tipe delegate diatas akan menjadi :

```
Func<Mahasiswa, bool> filter = x => x.IPK > 3.1;
Func<Mahasiswa, string> seleksi = x => x.Nama;
IEnumerable<string> hasil = kumpulanMahasiswa.
    Where(filter).
    Select(seleksi);
```

Kompiler C# dan VB 9.0 secara otomatis akan mentranslasikan bentuk query awal kita menjadi bentuk query yang kedua. Kemudian jika anda perhatikan bahwa prosedur Where dan Select telah menjadi kata kunci pada C# dan VB. Sehingga tentunya untuk melakukan query kita tidak perlu lagi memanggil secara langsung prosedur SQO namun dapat langsung menggunakan kata kunci seperti query yang pertama. Namun perlu diingat bahwa tidak semua prosedur SQO dijadikan kata kunci sehingga terkadang kita perlu secara langsung memanggil prosedur SOQ.

Sintaks LINQ secara komplit

Jika pada bagian sebelumnya kita telah mempelajari sebuah query yang sederhana terhadap sekumpulan objek namun sintaks query dari LINQ tentunya lebih luas dan kompleks dari sekedar query diatas. Maka disini sama-sama kita akan membahasnya secara mendalam.

Setiap query dimulai dari klausa from yang kemudian diakhiri dengan klausa select atau group. Alasan dari LINQ untuk mendahulukan klausa from ketimbang select berkaitan dengan kebutuhan dari fitur Intellisense pada Visual Studio. Karena untuk menggunakan fitur tersebut tentunya harus diketahui dahulu objek yang akan dioperasikan dan hal ini diperoleh dari klausa from. Klausa select akan memproyeksikan hasil dari ekspresi menjadi sebuah objek enumerable. Klausa group akan memproyeksikan hasil expresi menjadi sekumpulan group yang dibagi-bagi berdasarkan kondisi tertentu. Kode dibawah ini menunjukkan sintaks dari LINQ secara keseluruhan :

```

query-body ::=

join-clause*
(from-clause join-clause* | let-clause | where-clause)*
orderby-clause?
(select-clause | groupby-clause)
    query-continuation?

from-clause ::= from itemName in srcExpr

select-clause ::= select selExpr

groupby-clause ::= group selExpr by keyExpr

```

klausa from yang pertama dapat diikuti oleh satu atau lebih klausa from,let atau where. Klausa let akan memberikan nama terhadap hasil dari sebuah ekspresi, sedangkan klausa where akan mendefinisikan sebuah filter yang akan diaplikasikan terhadap sebuah item dalam hasil query. Setiap klausa from adalah sebuah generator yang mewakili iterasi terhadap sekumpulan objek. Sebuah klausa from dapat diikuti oleh klausa join. Dan kemudian klausa select atau group dapat didahului oleh kalusa orderby yang akan mempengaruhi cara pengurutan akhir dari elemen.

```

join-clause ::=
join itemName in srcExpr on keyExpr equals keyExpr
(into itemName)? 

orderby-clause ::= orderby (keyExpr (ascending | descending)?) *

query-continuation ::= into itemName query-body

```

Standard Query Operators

Seperti telah disinggung pada beberapa bagian sebelumnya LINQ memiliki sekumpulan API yang memiliki nama Standard Query Operators (SQO). API ini lah yang memberikan kemampuan LINQ untuk melakukan pengaksesan data mirip dengan yang biasa dilakukan dengan menggunakan Query SQL. SQO ini adalah kumpulan dari extension methods yang memperluas kelas mengimplementasikan interface `IEnumerable<T>`. Kemudian yang menarik adalah kebanyakan kelas kontainer dalam .NET framework mengimplementasikan interface `IEnumerable<T>` yang berarti kita tidak terlalu banyak merubah kode kita yang terdahulu untuk dapat menggunakan SQO. Dalam buku ini ada lima kategori dari SQO yang akan dibahas khusus yaitu `Projection, Restriction, Ordering, Grouping` dan `Aggregate`

Projection

`Projection` adalah kategori dari prosedur dalam SQO yang akan mengembalikan sekumpulan elemen yang dihasilkan dari proses pemilihan atau pembuatan sekumpulan elemen baru dari sekumpulan data masukan. Data yang diperoleh pada kumpulan keluaran bisa saja berbeda dengan data yang menjadi masukan

Select

Sama seperti kata SELECT pada SQL maka prosedur select ini akan menspesifikasikan kriteria dari elemen-elemen yang akan dipilih. Kemudian prosedur ini pun telah menjadi kata kunci pada C# 3.0. Prosedur ini memiliki 2 buah prototype yaitu :

```
public static IEnumerable<S> Select <T,S>(this IEnumerable<T> source,  
Func<T,S> selector)  
public static IEnumerable<S> Select <T,S>(this IEnumerable<T> source,  
Func<T,int> selector)
```

prototype yang pertama ini akan mengambil sebuah type delegate yang kemudian tentunya akan memanggil fungsi yang akan menerima masukan sekumpulan input(T) dan kemudian menghasilkan sekumpulan keluaran(S). Contoh penggunaanya adalah pada kutipan kode dibawah ini :

```
List<int> kumpulan = new List<int>() { 5, 2, 3 };  
var hasil = kumpulan.Select(x => x);  
foreach (var c in hasil)  
    Console.WriteLine(c);
```

Kode diatas akan menuliskan hasil “523” pada layar, hal ini dikarenakan masukan tidak mengalami perubahan, namun jika anda mengganti kode ($x \Rightarrow x$) menjadi ($x \Rightarrow 5$) maka tentunya hasil yang akan dituliskan dilayar adalah “555”.

Prototype yang kedua serupa dengan yang pertama perbedaannya terletak pada ditambahnya parameter pada type delegatanya. Fungsi yang akan dipanggil akan ditambahkan sebuah index dari elemen yang sedang diproses pada kumpulan masukan. Contoh penggunaanya adalah pada kode dibawah ini:

```
List<int> kumpulan = new List<int>() { 5, 2, 3 };  
var hasil = kumpulan.Select((x,y) => y);  
foreach (var c in hasil)  
    Console.WriteLine(c);
```

Kode diatas akan menuliskan hasil “123” pada layar, hal ini dikarenakan keluaran diset dengan nilai posisi dari elemen masukan. Jika anda berekperimen dengan mengganti kode ($x,y \Rightarrow x+y$) maka hasil yang akan dituliskan dilayar adalah “523” karena setiap elemen keluaran merupakan hasil pertambahan nilai elemen masukan dan nilai posisi dari elemen tersebut.

SelectMany

Prosedur ini digunakan untuk membuat sebuah proyeksi satu ke banyak dari setiap input masukan. Jika pada Select maka setiap satu elemen masukan akan ada satu keluaran sedangkan untuk SelectMany maka setiap ada satu masukan maka akan diproyeksikan menjadi sekumpulan elemen. Prototipe untuk prosedur ini ada dua buah yaitu

```
public static IEnumerable<S> SelectMany<T, S>(this IEnumerable<T> source,
Func<T, IEnumerable<S>> selector)
```

```
public static IEnumerable<S> SelectMany<T, S>(this IEnumerable<T> source,
Func<T, int, IEnumerable<S>> selector)
```

prototype yang pertama ini akan mengambil sebuah type delegate yang kemudian tentunya akan memanggil fungsi yang akan menerima masukan sekumpulan input(T) dan kemudian menghasilkan sekumpulan keluaran(IEnumerable<S>). Perlu diperhatikan bahwa keluaran dari SelectMany ini bukan merupakan elemen (S) seperti pada Select melainkan sebuah kumpulan elemen. Contoh penggunaanya adalah pada kutipan kode dibawah ini :

```
List<string> kumpulan = new List<string>()
{ "satu", "dua" };
var hasil = kumpulan.SelectMany((x=>x));
foreach (var c in hasil)
    Console.WriteLine(c + "|");
```

Kode diatas akan menuliskan hasil sebagai berikut “s|a|t|u|d|u|a|” pada layar, hal ini dikarenakan masukan tidak mengalami perubahan dan setiap string masukan akan dijadikan sebuah IEnumerable<char>, jika anda ganti SelectMany menjadi Select tentunya hasilnya adalah “satu|dua|”.

Prototype yang kedua serupa dengan yang pertama perbedaannya terletak pada ditambahnya parameter pada type delegatenya. Fungsi yang akan dipanggil akan ditambahkan sebuah index dari elemen yang sedang diproses pada kumpulan masukan. Contoh penggunaanya adalah pada kode dibawah ini:

```
List<string> kumpulan = new List<string>() { "satu", "dua" };
var hasil = kumpulan.SelectMany((x, y)=>y);
foreach (var c in hasil)
    Console.WriteLine(c + "|");
```

Kode diatas akan menuliskan hasil “0|1” pada layar, hal ini dikarenakan keluaran diset dengan nilai posisi dari elemen masukan. Jika anda berekperimen dengan mengganti kode $(x,y) \Rightarrow x+y$ maka hasil yang akan dituliskan dilayar adalah “s|a|t|u|1|d|u|a|2” karena setiap elemen keluaran dihasilkan dari merupakan hasil pertambahan nilai elemen masukan dan nilai posisi dari elemen tersebut kemudian dijadikan kumpulan char.

Contoh penggunaan dari SelectMany yang paling umum dipakai adalah untuk menirukan operasi join pada query SQL, meskipun dalam SQO pun ada prosedur join. (Silahkan lihat [lampiran B](#) untuk contoh kode pembuatan kelas Pegawai dan Jabatan)Contoh kodennya dibawah ini :

```
var pel = Pegawai.createPegawai();
var jab = Jabatan.createJabatan();
var hasil = jab.Where(x => x.ID == 1).
    SelectMany(
        x=>pel
        .Where(j=>j.IDJabatan ==x.ID)
        .Select(y=>new { Nama=y.Nama, NamaJabatan=x.NamaJabatan })
    );
foreach (var c in hasil)
    Console.WriteLine(c);
```

Dari potongan kode diatas terlihat bahwa sedang dilakukan proses join antara kumpulan objek Pegawai dengan kumpulan objek Jabatan. Pertama-tama dilakukan seleksi terhadap kumpulan jabatan untuk mengambil jabatan dengan nilai ID sama dengan 1. Setelah itu dengan prosedur SelectMany maka untuk setiap objek Jabatan yang memenuhi syarat akan dicari kumpulan objek Pegawai yang memiliki atribut ID yang sama dengan objek Jabatan tersebut dan hasilnya dikembalikan menjadi sekumpulan koleksi objek (<I Enumerable>). Jika anda berekperimen dengan mengganti operasi SelectMany menjadi Select maka hasil tidak akan menjadi kumpulan koleksi objek melainkan sebuah elemen List<T>

Restriction

Where

Sama seperti kata WHERE dalam Query SQL dan juga telah menjadi kata kunci dalam C# 3.0, prosedur ini pun akan memfilter sekumpulan objek dan mengembalikan kumpulan objek yang memenuhi syarat. Prototype untuk Where ada 2 yaitu :

```
public static I Enumerable<T> Where<T>(this I Enumerable<T> source, Func<T, bool> predicate);
public static I Enumerable<T> Where<T>(this I Enumerable<T> source, Func<T, int, bool> predicate);
```

prototype yang pertama ini akan mengambil sebuah type delegate yang kemudian tentunya akan memanggil fungsi yang akan menerima masukan sekumpulan input(T) yang kemudian melakukan seleksi terhadap masing-masing elemen pada T untuk menghasilkan sekumpulan keluaran(S) yang sesuai. Contoh penggunaanya adalah pada kutipan kode dibawah ini :

```
var kumpulan = new List<int>() { 5, 2, 3 };
var hasil = kumpulan.Where(x => x < 4);
foreach (var c in hasil)
    Console.Write (c);
```

Terlihat dalam kode diatas dilakukan filter terhadap kumpulan angka(integer) dengan filter angka yang lebih kecil dari 4. Maka dilayar akan dituliskan hasil "23".

Prototype yang kedua serupa dengan yang pertama perbedaannya terletak pada ditambahnya parameter pada type delegatenya. Fungsi yang akan dipanggil akan ditambahkan sebuah index dari elemen yang sedang diproses pada kumpulan masukan. Contoh penggunaannya adalah pada kode dibawah ini:

```
var p = new List<int>() { 5, 3, 1};
var hasil = p.Where((x,y) => x < y);
foreach (var c in hasil)
    Console.WriteLine(c);
```

Potongan kode diatas akan menuliskan "1" dilayar, hal ini karena posisi elemen integer 1 ada di urutan ke 3 dan tentunya nomor urutannya lebih besar dari nilai elemen tersebut.

Ordering

Ordering merupakan kumpulan operator yang akan melakukan operasi pengurutan terhadap koleksi objek.

OrderBy

Sama seperti ORDER BY dalam Query SQL maka prosedur ini pun akan mengurutkan kumpulan elemen berdasarkan nilai dari sebuah atribut dari elemen tersebut yang dijadikan kunci. Prototype untuk prosedur ini ada dua buah yaitu :

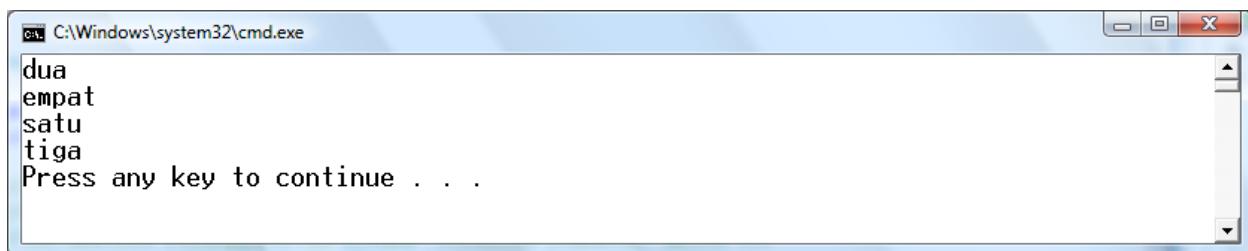
```
public static IOrderedEnumerable<TSource> OrderByDescending<TSource,
TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector);

public static IOrderedEnumerable<TSource> OrderByDescending<TSource,
TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector,
IComparer<TKey> comparer);
```

Prototype yang pertama akan menerima masukan fungsi yang menentukan properti dari elemen yang akan dijadikan kunci dalam pengurutan. Contoh kode penerapannya adalah :

```
var p = new List<string>() { "satu", "dua", "tiga", "empat" };
var hasil = p.OrderBy(x => x);
foreach (var c in hasil)
    Console.WriteLine(c);
```

Kode diatas akan menuliskan hasil di layar sebagai berikut :



The screenshot shows a Windows command prompt window titled 'cmd C:\Windows\system32\cmd.exe'. The window contains the following text:
dua
empat
satu
tiga
Press any key to continue . . .

Dari hasil diatas terlihat bahwa pembanding standar akan digunakan untuk melakukan pengurutan terhadap elemen yang ada dalam koleksi. Jika anda ingin menggunakan sebuah pembanding yang berbeda maka anda dapat menggunakan prototype yang kedua. Kemudian membuat sebuah kelas yang mengimplementasikan `IComparer<T>` seperti dibawah ini :

```
public class Pembanding : IComparer<string>
{
    public int Compare(string x, string y)
    {
        int result;
        if (x.Length > y.Length)
            result = 1;
        else if (x.Length < y.Length)
            result = -1;
        else
```

```

        result = 0;
    return result;
}

}

```

Kelas diatas akan melakukan perbandingan terhadap dua buah string berdasarkan panjang dari string tersebut. Prosedur compare yang merupakan kontrak dari interface IComparer akan mengembalikan nilai positif jika string pertama panjangnya lebih besar dari string kedua dan seterusnya. Fungsi ini lah yang akan dipanggil untuk membandingkan nilai besar kedua buah objek string.

Penggunaan IComparer dalam melakukan pengurutan terlihat pada kode dibawah ini :

```

var p = new List<string>() { "satu", "dua", "tiga", "empat" };
var hasil = p.OrderBy(x => x, new Pembanding());
foreach (var c in hasil)
    Console.WriteLine(c);

```

Kode diatas akan menghasilkan hasil sebagai berikut

```

C:\Windows\system32\cmd.exe
dua
satu
tiga
empat
Press any key to continue . . .

```

Terlihat dalam hasil bahwa string yang paling pendek ada di urutan teratas. Hal ini tentunya berbeda dengan penggunaan prototype yang pertama yang mengurutkan string berdasarkan karakter pertama pada string tersebut.

OrderByDescending

Memiliki fungsionalitas yang hampir sama dengan OrderBy hanya perbedaannya jika pada OrderBy elemen yang nilai keynya paling besar akan ada dalam urutan pertama dalam koleksi maka dalam OrderByDescending elemen yang nilainya paling kecil lah yang akan ada di urutan pertama dalam koleksi. Sama seperti OrderBy prosedur ini pun memiliki dua buah property yaitu :

```

public static IOrderedEnumerable<TSource> OrderByDescending<TSource,
TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector);

public static IOrderedEnumerable<TSource> OrderByDescending<TSource,
TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector,
IComparer<TKey> comparer);

```

Penggunaan kedua buah prototype ini sama dengan penggunaan OrderBy dan seperti telah dijelaskan sebelumnya perbedaanya hanyalah pada pengurutan koleksi elemen.

Reverse

Prosedur ini akan membalik urutan dari koleksi elemen. Prosedur ini hanya memiliki 1 buah prototype yaitu :

```
public static IEnumerable<TSource> Reverse<TSource>(this  
    IEnumerable<TSource> source);
```

contoh kode penggunaannya adalah :

```
var p = new List<string>() {"satu", "dua", "tiga", "empat" };  
var hasil = p.Select(x => x).Reverse();  
foreach (var c in hasil)  
    Console.WriteLine(c);
```

Hasil dari kode jika dieksekusi adalah



```
empat  
tiga  
dua  
satu  
Press any key to continue . . .
```

Terlihat dari hasilnya bahwa urutan dari koleksi telah dibalik

ThenBy

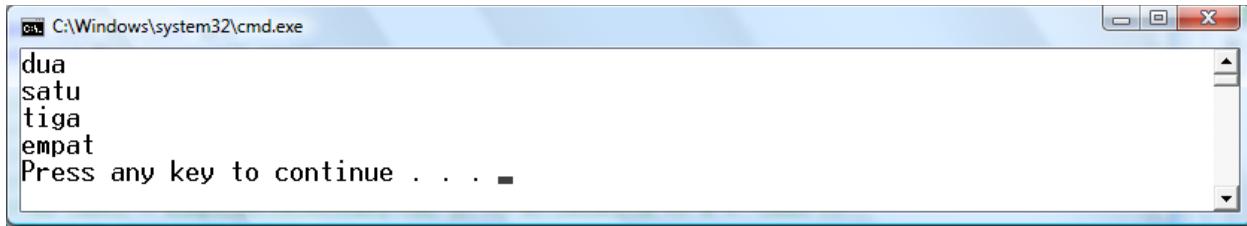
ThenBy ini berguna apabila kita ingin mengurutkan koleksi kita berdasarkan atas 2 kunci atau lebih. Karena seperti telah dilihat pada bagian sebelumnya prosedur OrderBy hanya menerima 1 kunci jika kita ingin menggunakan 2 kunci atau lebih maka kunci selanjutnya harus didefinisikan melalui prosedur ini. Sama seperti OrderBy maka prosedur ini memiliki 2 buah prototype yaitu :

```
public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>(this  
    IOrderedEnumerable<TSource> source, Func<TSource, TKey> keySelector);  
  
public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>(this  
    IOrderedEnumerable<TSource> source, Func<TSource, TKey> keySelector,  
    IComparer<TKey> comparer);
```

Prototype yang pertama akan menerima masukan fungsi yang menentukan properti dari elemen yang akan dijadikan kunci dalam pengurutan. Contoh kode penerapannya adalah :

```
var p = new List<string>() {"satu", "dua", "tiga", "empat" };  
var hasil = p.OrderBy(x => x.Length).ThenBy(x => x[0]);  
foreach (var c in hasil)  
    Console.WriteLine(c);
```

Kode diatas akan menuliskan hasil di layar sebagai berikut :



```
dua
satu
tiga
empat
Press any key to continue . . . -
```

Terlihat bahwa kode tersebut akan melakukan pengurutan pertama-tama berdasarkan panjang dari string masukan untuk kemudian untuk string yang panjangnya sama akan dibandingkan karakter pertamanya.

Sama seperti OrderBy maka prosedur ThenBy pun memiliki sebuah prototype yang akan menerima kelas yang mengimplementasikan interface `IComparer<T>` untuk menjadi pembanding.

ThenByDescending

Memiliki fungsionalitas yang hampir sama dengan ThenBy hanya perbedaannya jika pada ThenBy elemen yang nilai keynya paling besar akan ada dalam urutan pertama dalam koleksi maka dalam ThenByDescending

elemen yang nilainya paling kecil lah yang akan ada di urutan pertama dalam koleksi. Sama seperti ThenBy prosedur ini pun memiliki dua buah property yaitu :

```
public static IObservable<TSource> ThenByDescending<TSource,
TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector);

public static IObservable<TSource> ThenByDescending<TSource,
TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector,
IComparer<TKey> comparer);
```

Penggunaan kedua buah prototype ini sama dengan penggunaan ByDescending dan seperti telah dijelaskan sebelumnya perbedaanya hanyalah pada pengurutan koleksi elemen.

Grouping

Grouping merupakan kumpulan operator yang akan melakukan operasi penggabungan terhadap kumpulan dari koleksi objek.

Join

Prosedur ini akan melakukan operasi penggabungan antara 2 koleksi berdasarkan kesamaan antara kunci yang diperoleh dari setiap elemen yang ada dalam 2 koleksi tersebut. Prototype untuk Join ada dua buah yaitu :

```

public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(this
    IEnumerable<TOuter> outer, IEnumerable<TInner> inner, Func<TOuter, TKey>
    outerKeySelector, Func<TInner, TKey> innerKeySelector, Func<TOuter, TInner,
    TResult> resultSelector);

public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(this
    IEnumerable<TOuter> outer, IEnumerable<TInner> inner, Func<TOuter, TKey>
    outerKeySelector, Func<TInner, TKey> innerKeySelector, Func<TOuter, TInner,
    TResult> resultSelector, IEqualityComparer< TKey> comparer);

```

Prototype yang pertama ini akan mengambil 5 buah parameter, parameter pertama menandakan koleksi “outer” yang akan akan digabungkan. Kemudian parameter kedua menandakan koleksi “inner” lalu parameter ketiga menyatakan kunci dari koleksi “outer” diikuti parameter ke 4 yang menyatakan kunci dari koleksi “inner”. Parameter terakhir merupakan fungsi yang akan membuat hasil dari operasi join ini. Contoh kode penggunaan prototype ini adalah :

```

var pel = Pegawai.createPegawai();
var jab = Jabatan.createJabatan();
var hasil = pel.Join(jab,
    x => x.IDJabatan,
    y => y.ID,
    (x, y) => new { Nama = x.Nama, nama = y>NamaJabatan });

```

Untuk melakukan kode diatas maka anda perlu membuat sebuah kelas Pegawai dan Jabatan yang bisa anda lihat pada LAMPIRAN B. Kode diatas akan menghasilkan hasil sebagai berikut :

```

{ Nama = Ronald, nama = MIC Lead }
{ Nama = Fuady, nama = MIC Lead }
{ Nama = Zeddy, nama = DPE Team }
{ Nama = Narendra, nama = DPE Team }
{ Nama = Umar, nama = DPE Team }
{ Nama = Arief, nama = MIC Crew }
{ Nama = Fajar, nama = MIC Infrastructure Architect }
{ Nama = Anggriawan, nama = MIC Crew }
Press any key to continue . .

```

Terlihat dalam kode diatas bahwa penggabungan antara koleksi objek Pegawai dan Jabatan menggunakan kesamaan yang ada pada properti IDJabatan di objek Pegawai dengan properti ID di objek Jabatan.

Prototype yang kedua berguna jika anda ingin menggunakan bentuk komparasi kesamaan yang berbeda. Terlihat dalam contoh sebelumnya bahwa kesamaan antara IDJabatan dengan ID diperoleh melalui kesamaan antara nilai integer. Namun andaikan kuncinya berupa string dan anda menyatakan bahwa kunci tersebut cocok jika panjangnya sama atau jika kedua string itu sama jika tidak memperhatikan huruf besar atau kecilnya maka anda harus menggunakan prototype yang kedua ini dan membuat

sebuah kelas yang mengimplementasikan Interface `IEqualityComparer<T>`. Contoh kelas yang akan membandingkan 2 buah string tanpa melihat huruf besar atau kecilnya (case insensitive) :

```
public class PembandingKesamaan : IEqualityComparer<string>
{
    private CaseInsensitiveComparer myComparer;

    public PembandingKesamaan()
    {
        myComparer = new CaseInsensitiveComparer();
    }

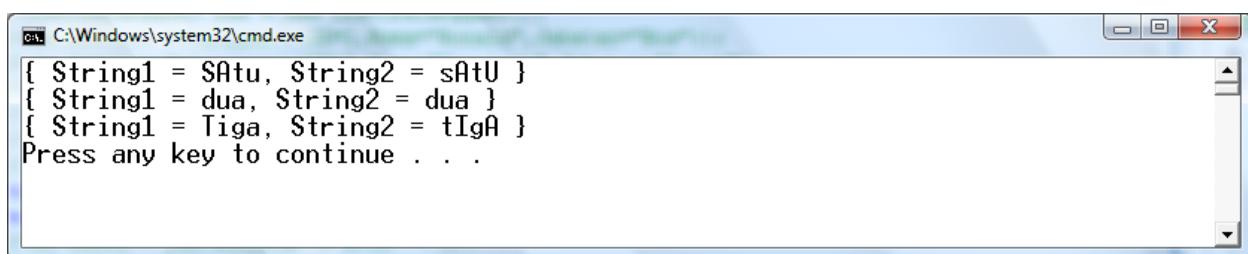
    public bool Equals(string x, string y)
    {
        bool hasil;
        if (myComparer.Compare(x, y) == 0)
        {
            hasil = true;
        }
        else
        {
            hasil = false;
        }
        return hasil;
    }

    public int GetHashCode(string obj)
    {
        return obj.ToString().ToLower().GetHashCode();
    }
}
```

Kelas diatas akan menghasilkan hasil kemudian untuk setiap string yang sama tanpa memperhatikan huruf besar atau kecilnya. Objek dari kelas ini lah yang akan digunakan sebagai pembanding pada parameter ke 6 di prototype kedua sebagai berikut :

```
var KoleksiPertama = new List<string> { "SAtu", "dua", "Tiga" };
var KoleksiKedua = new List<string> { "sAtU", "dua", "tIgA" };
var hasil = KoleksiPertama.Join(KoleksiKedua,
    x => x,
    y => y,
    (x, y) => new { String1 = x, String2 = y },
    new PembandingKesamaan());
```

hasil dari kode diatas adalah :

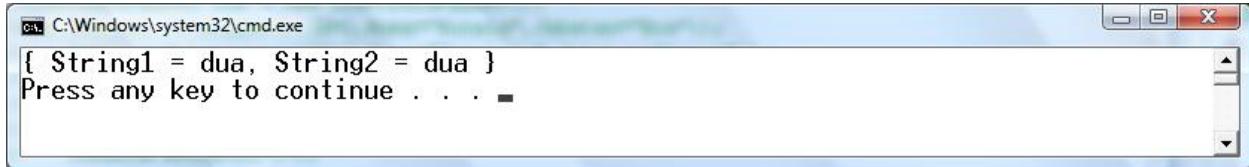


The screenshot shows a Windows Command Prompt window titled 'cmd C:\Windows\system32\cmd.exe'. The output of the code execution is displayed in the window:

```
{ String1 = SAtu, String2 = sAtU }
{ String1 = dua, String2 = dua }
{ String1 = Tiga, String2 = tIgA }
```

Press any key to continue . . .

Terlihat bahwa string "SAtu" dan "sAtU" dianggap sama karena jika kita tidak melihatnya secara case insensitive maka kedua string tersebut akan dianggap sama. Jika anda mengambil experiment dengan menghapus parameter terakhir atau dengan kata lain menggunakan prototype pertama maka hasilnya adalah :



```
{ String1 = dua, String2 = dua }
Press any key to continue . . . -
```

Hal ini terjadi karena secara standar LINQ akan memanggil komparator kesamaan standar dari string yang yang memiliki sifat case sensitive. Sebagai informasi jika operasi kesamaan dilakukan terhadap objek maka secara standar nilai yang akan dianalisa adalah nilai hashcodenya yang diperoleh dari fungsi GetHashCode yang ada pada setiap objek.

GroupJoin

Prosedur ini mirip seperti join yaitu akan menggabungkan dua buah kumpulan objek berdasarkan nilai property yang menjadi kuncinya. Namun berbeda dengan join yang akan mengirimkan sebuah elemen inner dan elemen outer ke dalam resultSelectornya maka pada GroupJoin semua elemen pada kelompok inner yang cocok dengan sebuah elemen outer yang akan dikirimkan ke resultSelector. Sehingga efeknya adalah prosedur resultSelector hanya akan dipanggil sekali untuk setiap elemen yang ada di koleksi outer. Prototypenya ada dua buah dengan perbedaan pada parameter terakhir yang berupa objek yang akan melakukan komparasi.

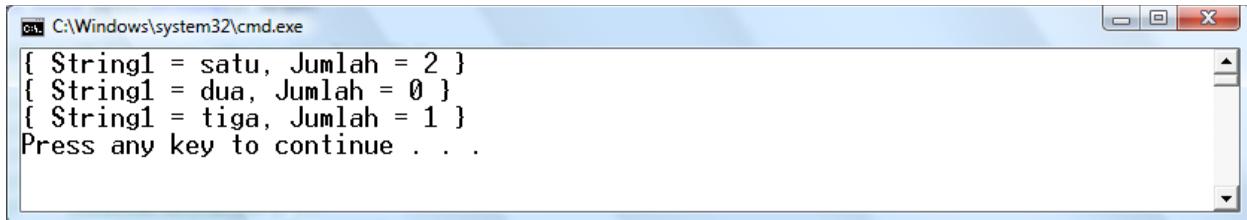
```
public static IEnumerable<TResult> GroupJoin<TOuter, TInner, TKey,
TResult>(this IEnumerable<TOuter> outer, IEnumerable<TInner> inner,
Func<TOuter, TKey> outerKeySelector, Func<TInner, TKey> innerKeySelector,
Func<TOuter, IEnumerable<TInner>, TResult> resultSelector);

public static IEqualityComparer<TKey> Comparer<TKey> Comparer();
public static IEqualityComparer<TKey> Comparer(IEqualityComparer<TKey> comparer);
```

Contoh kode penggunaan prototype yang pertama adalah kode dibawah ini :

```
var KoleksiPertama = new List<string> { "satu", "dua", "tiga" };
var KoleksiKedua = new List<string> { "satu", "satu", "tiga" };
var hasil = KoleksiPertama.GroupJoin(KoleksiKedua,
    x => x,
    y => y,
    (x, y) => new { String1 = x, String2 = y.Count() } );
```

Kode diatas akan berusaha untuk menghitung jumlah elemen yang sama di koleksi inner dengan sebuah elemen di koleksi outer. Kode diatas akan menghasilkan hasil sebagai berikut ketika dieksekusi :



```
C:\Windows\system32\cmd.exe
{ String1 = satu, Jumlah = 2 }
{ String1 = dua, Jumlah = 0 }
{ String1 = tiga, Jumlah = 1 }
Press any key to continue . . .
```

Terlihat dari hasil yang dihasilkan bahwa untuk elemen string dengan nilai "satu" pada kelompok outer ada dua buah elemen di kelompok inner. Yang menarik adalah meskipun elemen dengan nilai "dua" tidak ada dalam kelompok inner tapi karena ada dalam outer maka akan masuk kedalam kelompok hasil, jika anda menggunakan join maka hal ini tidak akan terjadi karena prosedur join menggunakan sifat inner join yang sama seperti SQL pada umumnya.

Prototype yang kedua akan menerima parameter tambahan yang berupa kelas yang akan melakukan perbandingan terhadap kedua key dari koleksi inner dan outer. Kode dibawah ini akan menghasilkan hasil yang sama dengan kode diatas mengingat kelas digunakan akan membandingkan objek string secara case insensitive.

```
var KoleksiPertama = new List<string> { "satu", "dua", "tiga" };
var KoleksiKedua = new List<string> { "saTU", "sAtu", "tiga" };
var hasil = KoleksiPertama.GroupJoin(KoleksiKedua,
    x => x,
    y => y,
    (x, y) => new { String1 = x, Jumlah = y.Count() }
    , new PembandingKesamaan());
```

Aggregate

Aggregate merupakan sekumpulan prosedur SQO yang akan melakukan operasi aggregasi terhadap elemen pada kumpulan masukan. Berbeda dengan kategori-kategori dalam SQO sebelumnya yang mengembalikan sekumpulan objek dalam `IEnumerable<T>`, Aggregate hanya akan mengembalikan sebuah objek yang merupakan hasil agregasi dari setiap objek yang ada dalam koleksi masukan.

Average

Merupakan prosedur yang akan melakukan operasi pencarian nilai rata-rata dari properti nilai numerik pada sekumpulan objek. Prototype dari prosedur ini relatif banyak yaitu ada 20 buah yaitu :

```
public static decimal? Average(this IEnumerable<decimal?> source);

public static decimal Average(this IEnumerable<decimal> source);

public static double? Average(this IEnumerable<double?> source);

public static double Average(this IEnumerable<double> source);

public static float? Average(this IEnumerable<float?> source);

public static float Average(this IEnumerable<float> source);

public static double? Average(this IEnumerable<int?> source);

public static double Average(this IEnumerable<int> source);
```

```

public static double? Average(this IEnumerable<long?> source);

public static double Average(this IEnumerable<long> source);

public static decimal? Average<TSource>(this IEnumerable<TSource> source,
Func<TSource, decimal?> selector);

public static decimal Average<TSource>(this IEnumerable<TSource> source,
Func<TSource, decimal> selector);

public static double? Average<TSource>(this IEnumerable<TSource> source,
Func<TSource, double?> selector);

public static double Average<TSource>(this IEnumerable<TSource> source,
Func<TSource, double> selector);

public static float? Average<TSource>(this IEnumerable<TSource> source,
Func<TSource, float?> selector);

public static float Average<TSource>(this IEnumerable<TSource> source,
Func<TSource, float> selector);

public static double? Average<TSource>(this IEnumerable<TSource> source,
Func<TSource, int?> selector);

public static double Average<TSource>(this IEnumerable<TSource> source,
Func<TSource, int> selector);

public static double? Average<TSource>(this IEnumerable<TSource> source,
Func<TSource, long?> selector);

public static double Average<TSource>(this IEnumerable<TSource> source,
Func<TSource, long> selector);

```

Meskipun ada 20 prototype tapi sebenarnya perbedaannya hanya pada hasil kembalian dan juga jumlah parameter masukan. Contoh kode dibawah ini akan melakukan operasi rata-rata terhadap sekumpulan bilangan integer :

```

var koleksi = new List<int> { 1, 2, 4, 5, 6 };
    double hasil = koleksi.Average();
    Console.WriteLine(hasil);

```

Kegunaan dari tipe prototype yang menerima satu parameter tambahan adalah parameter tambahan tersebut akan menjadi selektor dari property yang ada dalam objek. Sebagai contoh misalnya kita ingin mencari jumlah rata-rata panjang dari sekumpulan string maka kita dapat menuliskan kode dibawah ini :

```

var koleksi = new List<string> { "satu", "dua", "tiga" };
var hasil = koleksi.Average(x=>x.Length);
Console.WriteLine(hasil);

```

Kode diatas akan memilih properti Length dari objek string untuk dimasukkan kedalam perhitungan nilai rata-rata.

Count

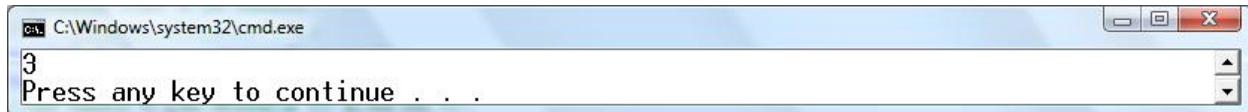
Merupakan prosedur yang akan melakukan operasi perhitungan terhadap banyaknya anggota dari sekumpulan objek. Prototype dari prosedur ini ada dua buah yaitu :

```
public static int Count<TSource>(this IEnumerable<TSource> source);  
  
public static int Count<TSource>(this IEnumerable<TSource> source,  
Func<TSource, bool> predicate);
```

prosedur yang pertama akan langsung menghitung jumlah objek yang ada dalam kumpulan objek. Penggunaannya ada dalam kode dibawah ini :

```
var koleksi = new List<string> { "satu", "dua", "tiga" };  
var hasil = koleksi.Count();  
Console.WriteLine(hasil);
```

hasil dari kode diatas adalah:



A screenshot of a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The window contains the text '3' followed by 'Press any key to continue . . .'. The window has standard Windows-style borders and title bar.

Sedangkan prototype ke dua memiliki sebuah parameter masukan yang akan memanggil sebuah tipe delegate yang akan melakukan filter terhadap setiap objek dalam kumpulan. Sehingga misalnya anda ingin menghitung jumlah string yang panjangnya lebih dari 3 dalam sekumpulan objek , tentunya anda akan membuat sebuah fungsi yang akan mengecek apakah jumlah karakter dari string tersebut lebih besar dari 3 atau tidak seperti pada kode dibawah ini:

```
var koleksi = new List<string> { "satu", "dua", "tiga" };  
var hasil = koleksi.Count(x => x.Length > 3);  
Console.WriteLine(hasil);
```

Kode diatas akan menghasilkan hasil sebagai berikut :



A screenshot of a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The window contains the text '2' followed by 'Press any key to continue . . . -'. The window has standard Windows-style borders and title bar.

LongCount

Merupakan prosedur yang akan melakukan operasi perhitungan terhadap jumlah anggota dari sekumpulan objek. Sama seperti Count namun perbedaanya ada pada tipe kembaliannya jika pada count kembaliannya berupa integer pada LongCount kembalian akan berupa Long. Prototype dari prosedur ini ada dua buah yaitu :

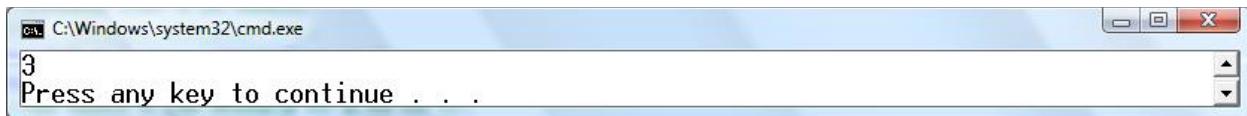
```
public static long Count<TSource>(this IEnumerable<TSource> source);

public static long Count<TSource>(this IEnumerable<TSource> source,
Func<TSource, bool> predicate);
```

prosedur yang pertama akan langsung menghitung jumlah objek yang ada dalam kumpulan objek. Penggunaannya ada dalam kode dibawah ini :

```
var koleksi = new List<string> { "satu", "dua", "tiga" };
var hasil = koleksi.LongCount();
Console.WriteLine(hasil);
```

hasil dari kode diatas adalah:



Sedangkan prototype ke dua memiliki sebuah parameter masukan yang akan memanggil sebuah tipe delegate yang akan melakukan filter terhadap setiap objek dalam kumpulan. Sehingga misalnya anda ingin menghitung jumlah string yang panjangnya lebih dari 3 dalam sekumpulan objek , tentunya anda akan membuat sebuah fungsi yang akan mengecek apakah jumlah karakter dari string tersebut lebih besar dari 3 atau tidak seperti pada kode dibawah ini:

```
var koleksi = new List<string> { "satu", "dua", "tiga" };
var hasil = koleksi.LongCount(x => x.Length > 3);
Console.WriteLine(hasil);
```

Kode diatas akan menghasilkan hasil sebagai berikut :



Max

Merupakan prosedur yang akan menghitung nilai maksimum dari sebuah properti dalam sekumpulan objek. Prototype untuk prosedur ini sama ada 20 buah yaitu :

```
public static decimal? Max(this IEnumerable<decimal?> source);

public static decimal Max(this IEnumerable<decimal> source);

public static double? Max(this IEnumerable<double?> source);

public static double Max(this IEnumerable<double> source);

public static float? Max(this IEnumerable<float?> source);

public static float Max(this IEnumerable<float> source);

public static int? Max(this IEnumerable<int?> source);
```

```

public static int Max(this IEnumerable<int> source);

public static long? Max(this IEnumerable<long?> source);

public static long Max(this IEnumerable<long> source);

public static TSource Max<TSource>(this IEnumerable<TSource> source);

public static decimal? Max<TSource>(this IEnumerable<TSource> source,
Func<TSource, decimal?> selector);

public static decimal Max<TSource>(this IEnumerable<TSource> source,
Func<TSource, decimal> selector);

public static double? Max<TSource>(this IEnumerable<TSource> source,
Func<TSource, double?> selector);

public static double Max<TSource>(this IEnumerable<TSource> source,
Func<TSource, double> selector);

public static float? Max<TSource>(this IEnumerable<TSource> source,
Func<TSource, float?> selector);

public static float Max<TSource>(this IEnumerable<TSource> source,
Func<TSource, float> selector);

public static int? Max<TSource>(this IEnumerable<TSource> source,
Func<TSource, int?> selector);

public static int Max<TSource>(this IEnumerable<TSource> source,
Func<TSource, int> selector);

public static long? Max<TSource>(this IEnumerable<TSource> source,
Func<TSource, long?> selector);

public static long Max<TSource>(this IEnumerable<TSource> source,
Func<TSource, long> selector);

public static TResult Max<TSource, TResult>(this IEnumerable<TSource> source,
Func<TSource, TResult> selector);

```

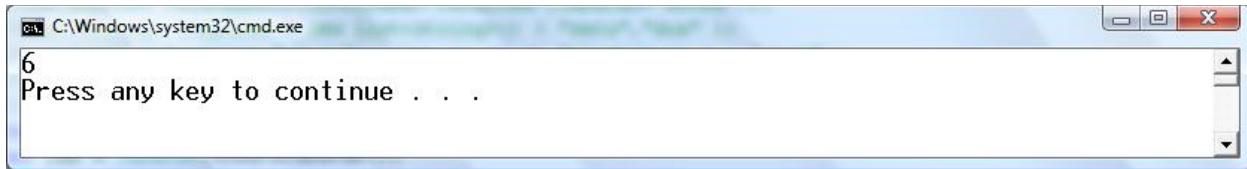
meskipun ada 22 prototype namun sebenarnya dapat digolongkan menjadi 4 prototype. Golongan yang pertama adalah yang melakukan operasi pencarian nilai maksimum terhadap kumpulan objek numerik seperti decimal,decimal?,double,double?,int,int?,float,float?. Contoh penggunaan dari prototype ini adalah :

```

var koleksi = new List<int> { 1,6,5,3};
var hasil = koleksi.Max();
Console.WriteLine(hasil);

```

Kode diatas akan mencari nilai maksimum dari kumpulan integer dan hasilnya adalah :

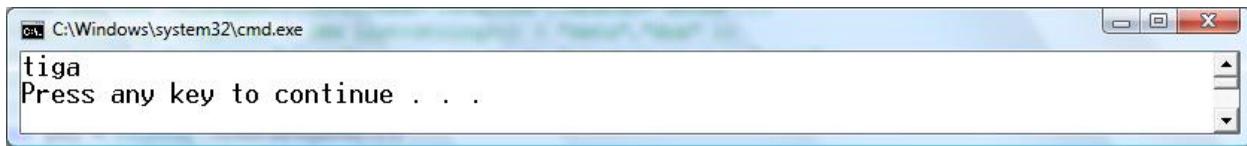


```
C:\Windows\system32\cmd.exe
6
Press any key to continue . . .
```

Golongan prototype yang kedua akan mencari nilai terbesar dari kumpulan objek yang bukan terdiri dari kumpulan numerik seperti dicontohkan pada kode dibawah ini :

```
var koleksi = new List<string> { "satu", "dua", "tiga", "empat" };
var hasil = koleksi.Max();
Console.WriteLine(hasil);
```

Kode diatas akan mencari nilai terbesar dari kumpulan string yang tentunya secara standar didasarkan pada urutan alphabet sehingga hasilnya ada sebagai berikut :

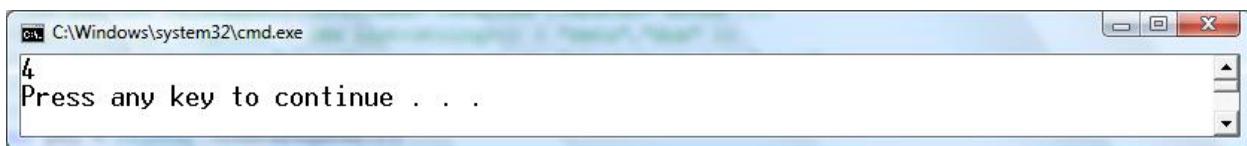


```
C:\Windows\system32\cmd.exe
tiga
Press any key to continue . . .
```

Sedangkan golongan prototype yang ketiga akan melakukan pencarian nilai maksimum terhadap sebuah properti dari objek yang ada dalam kumpulan. Properti yang didukung adalah kumpulan objek numerik seperti yang telah dijelaskan diatas. . Sehingga prototype ini akan meminta masukan sebuah tipe delegate yang akan memilih properti yang mana dari sebuah objek yang akan dicari nilai maksimumnya. Contoh kode penggunaanya adalah :

```
var koleksi = new List<string> { "satu", "dua", "tiga" };
var hasil = koleksi.Max(x=>x.Length);
Console.WriteLine(hasil);
```

Kode diatas akan mencari nilai maksimum panjang string yang ada dari sekumpulan objek string, hasil dari kode diatas adalah :



```
C:\Windows\system32\cmd.exe
4
Press any key to continue . . .
```

Tipe prototype yang keempat ini memiliki fungsi yang serupa dengan yang ketiga namun perbedaanya adalah bahwa property yang dapat dikenakan operasi max bukan merupakan elemen yang bertipe numerik, sebagai contoh dalam kode dibawah ini akan dilakukan operasi Max terhadap kumpulan objek yang mengambil property nilai substring –nya.

```
var koleksi = new List<string> { "satu", "dua", "tiga", "empat" };
var hasil = koleksi.Max(x=>x.Substring(1));
Console.WriteLine(hasil);
```

Dalam kode diatas hasil kembalian dari prosedur Max adalah sebuah string yang tanpa karakter pertamanya dengan urutan alphabet paling tinggi :



Min

Merupakan prosedur yang akan menghitung nilai minimum dari sebuah properti dalam sekumpulan objek. Prototype untuk prosedur ini sama ada 22 buah yaitu :

```
public static decimal? Min(this IEnumerable<decimal?> source);

public static decimal Min(this IEnumerable<decimal> source);

public static double? Min(this IEnumerable<double?> source);

public static double Min(this IEnumerable<double> source);

public static float? Min(this IEnumerable<float?> source);

public static float Min(this IEnumerable<float> source);

public static int? Min(this IEnumerable<int?> source);

public static int Min(this IEnumerable<int> source);

public static long? Min (this IEnumerable<long?> source);

public static long Min(this IEnumerable<long> source);

public static TSource Min<TSource>(this IEnumerable<TSource> source);

public static decimal? Min<TSource>(this IEnumerable<TSource> source,
Func<TSource, decimal?> selector);

public static decimal Min<TSource>(this IEnumerable<TSource> source,
Func<TSource, decimal> selector);

public static double? Min<TSource>(this IEnumerable<TSource> source,
Func<TSource, double?> selector);

public static double Min<TSource>(this IEnumerable<TSource> source,
Func<TSource, double> selector);

public static float? Min<TSource>(this IEnumerable<TSource> source,
Func<TSource, float?> selector);

public static float Min<TSource>(this IEnumerable<TSource> source,
Func<TSource, float> selector);

public static int? Min<TSource>(this IEnumerable<TSource> source,
Func<TSource, int?> selector);
```

```

public static int Min<TSource>(this IEnumerable<TSource> source,
Func<TSource, int> selector);

public static long? Min<TSource>(this IEnumerable<TSource> source,
Func<TSource, long?> selector);

public static long Min<TSource>(this IEnumerable<TSource> source,
Func<TSource, long> selector);

public static TResult Min<TSource, TResult>(this IEnumerable<TSource> source,
Func<TSource, TResult> selector);

```

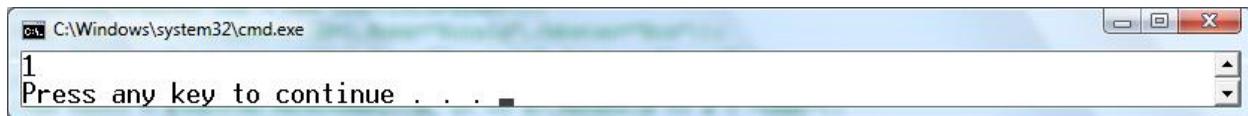
meskipun ada 22 prototype namun sebenarnya dapat digolongkan menjadi 4 prototype. Golongan yang pertama adalah yang melakukan operasi pencarian nilai minimum terhadap kumpulan objek numerik seperti decimal,decimal?,double,double?,int,int?,float,float?. Contoh penggunaan dari prototype ini adalah :

```

var koleksi = new List<int> { 1, 6, 5, 3};
var hasil = koleksi.Min();
Console.WriteLine(hasil);

```

Kode diatas akan mencari nilai minimum dari kumpulan integer dan hasilnya adalah :



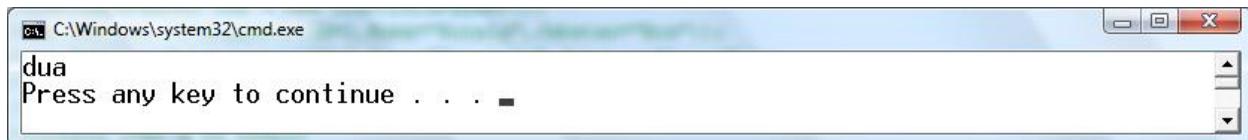
Golongan prototype yang kedua akan mencari nilai terkecil dari kumpulan objek yang bukan terdiri dari kumpulan numerik seperti dicontohkan pada kode dibawah ini :

```

var koleksi = new List<string> { "satu", "dua", "tiga", "empat" };
var hasil = koleksi.Min();
Console.WriteLine(hasil);

```

Kode diatas akan mencari nilai terkecil dari kumpulan string yang tentunya secara standar didasarkan pada urutan alphabet sehingga hasilnya ada sebagai berikut :



Sedangkan golongan prototype yang ketiga akan melakukan pencarian nilai minimum terhadap sebuah properti dari objek yang ada dalam kumpulan. Properti yang didukung adalah kumpulan objek numerik seperti yang telah dijelaskan diatas. Sehingga prototype ini akan meminta masukan sebuah tipe delegate yang akan memilih properti yang mana dari sebuah objek yang akan dicari nilai minimumnya. Contoh kode penggunaanya adalah :

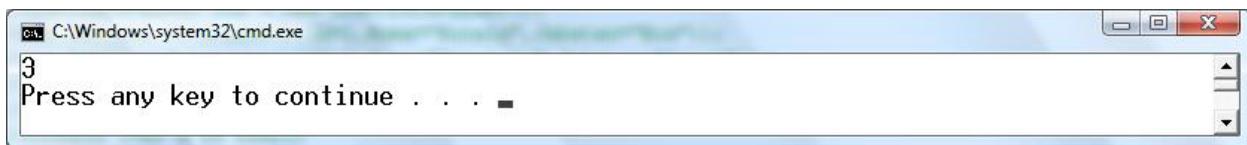
```

var koleksi = new List<string> { "satu", "dua", "tiga" };

```

```
var hasil = koleksi.Min(x=>x.Length);
Console.WriteLine(hasil);
```

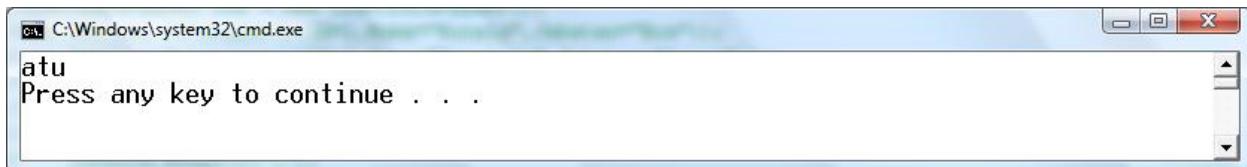
Kode diatas akan mencari nilai terkecil panjang string yang ada dari sekumpulan objek string, hasil dari kode diatas adalah :



Tipe prototype yang keempat ini memiliki fungsi yang serupa dengan yang ketiga namun perbedaanya adalah bahwa property yang dapat dikenakan operasi max bukan merupakan elemen yang bertipe numerik, sebagai contoh dalam kode dibawah ini akan dilakukan operasi Min terhadap kumpulan objek yang mengambil property nilai substring -nya.

```
var koleksi = new List<string> { "satu", "dua", "tiga", "empat" };
var hasil = koleksi.Min(x=>x.Substring(1));
Console.WriteLine(hasil);
```

Dalam kode diatas hasil kembalian dari prosedur Min adalah sebuah string yang tanpa karakter pertamanya dengan urutan alphabet paling rendah :



Sum

Merupakan prosedur yang akan melakukan operasi penjumlahan untuk nilai dari properti nilai numerik pada sekumpulan objek. Prototype dari prosedur ini relatif banyak yaitu ada 20 buah yaitu :

```
public static decimal? Sum(this IEnumerable<decimal?> source);

public static decimal Sum(this IEnumerable<decimal> source);

public static double? Sum(this IEnumerable<double?> source);

public static double Sum(this IEnumerable<double> source);

public static float? Sum(this IEnumerable<float?> source);

public static float Sum(this IEnumerable<float> source);

public static double? Sum(this IEnumerable<int?> source);

public static double Sum(this IEnumerable<int> source);

public static double? Sum(this IEnumerable<long?> source);
```

```

public static double Sum(this IEnumerable<long> source);

public static decimal? Sum<TSource>(this IEnumerable<TSource> source,
Func<TSource, decimal?> selector);

public static decimal Sum<TSource>(this IEnumerable<TSource> source,
Func<TSource, decimal> selector);

public static double? Sum<TSource>(this IEnumerable<TSource> source,
Func<TSource, double?> selector);

public static double Sum<TSource>(this IEnumerable<TSource> source,
Func<TSource, double> selector);

public static float? Sum<TSource>(this IEnumerable<TSource> source,
Func<TSource, float?> selector);

public static float Sum<TSource>(this IEnumerable<TSource> source,
Func<TSource, float> selector);

public static double? Sum<TSource>(this IEnumerable<TSource> source,
Func<TSource, int?> selector);

public static double Sum<TSource>(this IEnumerable<TSource> source,
Func<TSource, int> selector);

public static double? Sum<TSource>(this IEnumerable<TSource> source,
Func<TSource, long?> selector);

public static double Sum<TSource>(this IEnumerable<TSource> source,
Func<TSource, long> selector);

```

meskipun ada 20 prototype tapi sebenarnya perbedaannya hanya pada hasil kembalian dan juga jumlah parameter masukan. Contoh kode dibawah ini akan melakukan operasi pemjumlahan terhadap sekumpulan bilangan integer :

```

var koleksi = new List<int> { 1, 2, 4, 5, 6 };
double hasil = koleksi.Sum();
Console.WriteLine(hasil);

```

Kegunaan dari tipe prototype yang menerima satu parameter tambahan adalah parameter tambahan tersebut menjadi selektor dari property yang ada dalam objek. Sebagai contoh misalnya kita ingin mencari jumlah panjang dari sekumpulan string maka kita dapat menuliskan kode dibawah ini :

```

var koleksi = new List<string> { "satu", "dua", "tiga" };
var hasil = koleksi.Sum(x=>x.Length);
Console.WriteLine(hasil);

```

Kode diatas akan memilih properti Length dari objek string untuk dimasukkan kedalam perhitungan nilai penjumlahannya.

Aggregate

Merupakan prosedur yang akan melakukan operasi khusus terhadap kumpulan elemen. Maksud dari operasi khusus ini adalah operasi antar elemen yang tidak dispesifikasikan oleh SQO, sebagai contoh kita ingin melakukan algoritma fibonacci (fungsi rekursif yang akan menjumlahkan nilai elemen dengan hasil penjumlahan sebelumnya) terhadap kumpulan kita maka kita dapat menggunakan aggregate untuk melakukannya, hal ini pun akan ditunjukkan dalam contoh kode dalam buku ini. Prototype untuk prosedur ini ada 3 buah yaitu :

```
public static TSource Aggregate<TSource>(this IEnumerable<TSource> source,
Func<TSource, TSource, TSource> func);

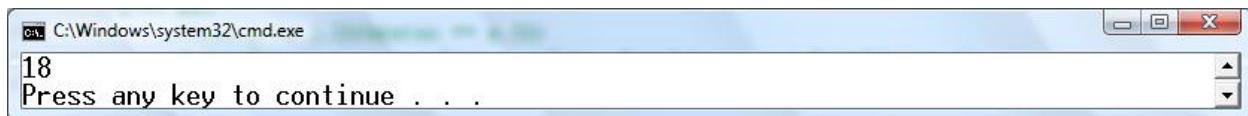
public static TAccumulate Aggregate<TSource, TAccumulate>(this
IEnumerable<TSource> source, TAccumulate seed, Func<TAccumulate, TSource,
TAccumulate> func);

public static TResult Aggregate<TSource, TAccumulate, TResult>(this
IEnumerable<TSource> source, TAccumulate seed, Func<TAccumulate, TSource,
TAccumulate> func, Func<TAccumulate, TResult> resultSelector);
```

Prototype yang pertama akan menerima masukan sebuah type delegate yang berisikan sebuah algoritma yang akan dioperasikan terhadap kedua elemen. Contoh kode yang akan melakukan operasi fibonacci adalah

```
var koleksi = new List<int> { 1, 2, 4, 5, 6 };
var hasil = koleksi.Aggregate((x, y) => x + y);
Console.WriteLine(hasil);
```

Terlihat dalam parameter masukan prosedur Aggregate penulis memasukkan sebuah lambda expresion yang akan menambahkan parameter pertama dan parameter kedua. Nilai dari parameter pertama adalah nilai dari hasil perhitungan sebelumnya, sehingga algoritma fibonacci dapat diselesaikan hanya dengan 1 baris kode. Hasil dari kode diatas adalah :



Prototype ke dua memiliki sebuah parameter tambahan yaitu seed dari operasi ini. Maksud dari seed adalah nilai awal yang akan digunakan dalam proses aggregasi sehingga jika kita menuliskan kode dibawah ini maka nilai angka 12 akan ikut dalam perhitungan sebagai elemen pertama :

```
var koleksi = new List<int> { 1, 2, 4, 5, 6 };
var hasil = koleksi.Aggregate(12, (x, y) => x + y);
Console.WriteLine(hasil);
```

Hasil dari kode diatas adalah :



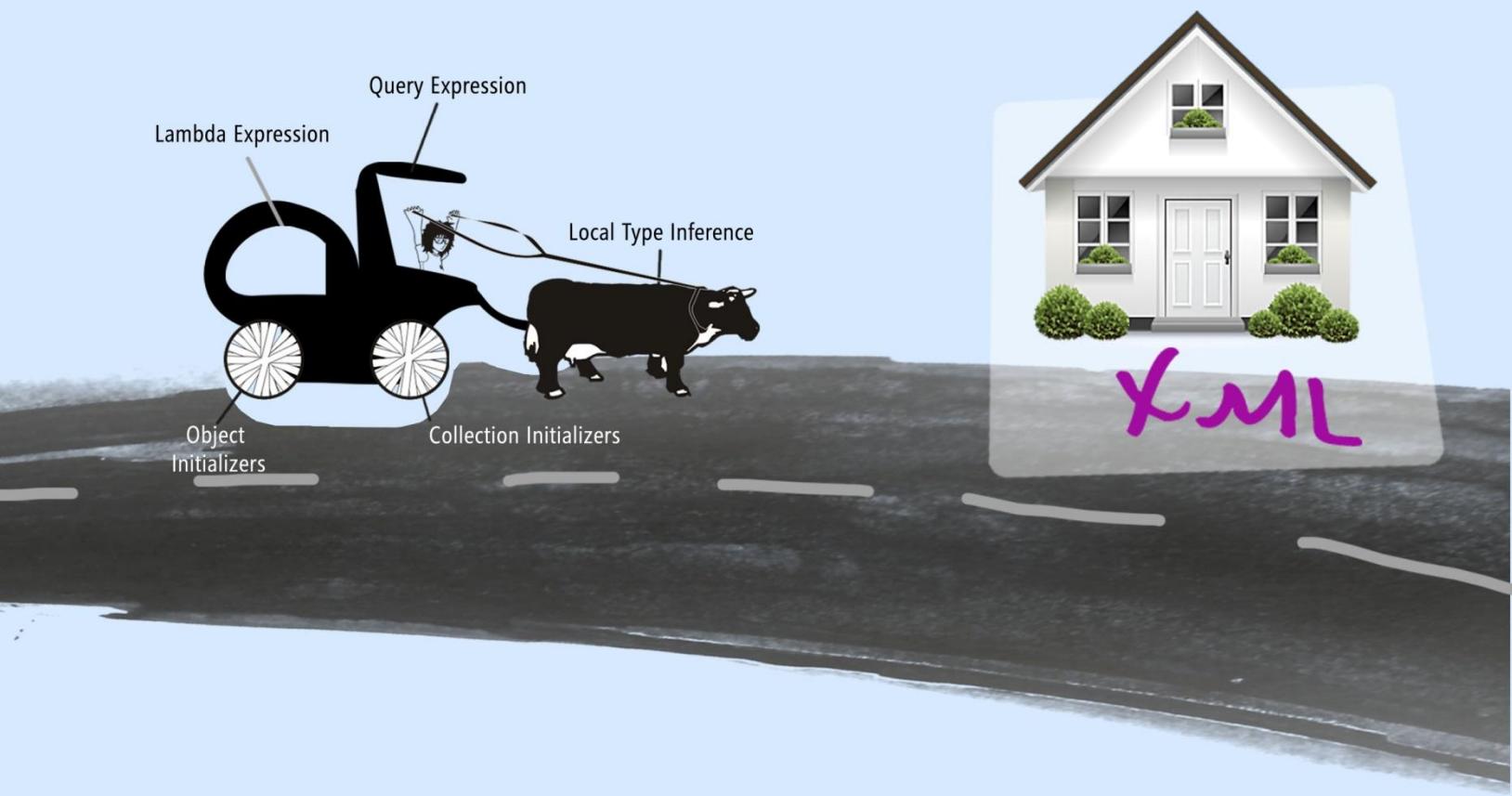
```
30
Press any key to continue . . .
```

Kemudian prototype terakhir menambahkan 1 lagi parameter yang merupakan type delegate. Parameter ini akan memanggil sebuah fungsi seleksi terhadap kumpulan elemen yang dihasilkan. Mirip seperti fungsi SELECT namun tentunya kita tidak dapat menggunakan SELECT terhadap hasil dari aggregate karena hasilnya bukan IEnumerate. Contoh kode dibawah ini akan merubah hasil yang berupa integer menjadi string :

```
var koleksi = new List<int> { 1, 2, 4, 5, 6 };
string hasil = koleksi.Aggregate(12, (x,
    y) => x + y,
    (x=> Convert.ToString(x)));
Console.WriteLine(hasil);
```

Penutup

LINQ to Object merupakan sebuah bagian dari LINQ yang sangat menarik. Dengan menggunakan LINQ to Object kita mendapatkan aroma deklaratif ketika sedang melakukan operasi terhadap objek. Sehingga seolah-olah objek kita disimpan dalam basis data. SQO pun menjadi fondasi yang penting ketika kita akan menggunakan LINQ untuk beroperasi terhadap data. Terlepas dari bentuk data apa yang akan kita operasikan, SQO akan tetap sangat berguna untuk melakukan Query.



LINQ to XML

LINQ to XML

Dalam pemberhentian yang kedua ini kita akan sama-sama melihat bagaimana kedigdayaan LINQ tetap berlanjut ketika berhadapan dengan XML di pondok LINQ to XML. XML seperti telah diketahui bersama merupakan salah satu format penyimpanan data yang sedang ramai digunakan di dunia industri. Namun dari sisi developer permrogramman terhadap XML bukanlah sesuatu hal yang mudah. DOM framework yang sering digunakan dalam operasi terhadap XML memerlukan kita untuk menulis banyak sekali kode hanya dalam rangka menghasilkan keluaran XML yang tidak terlalu banyak. Kemudian jika kita ingin melakukan operasi pencarian terhadap sebuah dokumen XML kita perlu menggunakan fitur DOM seperti XPath dan ini tidak sama seperti bahasa Query lainnya seperti SQL dan tentunya para developer harus mempelajarinya lagi.

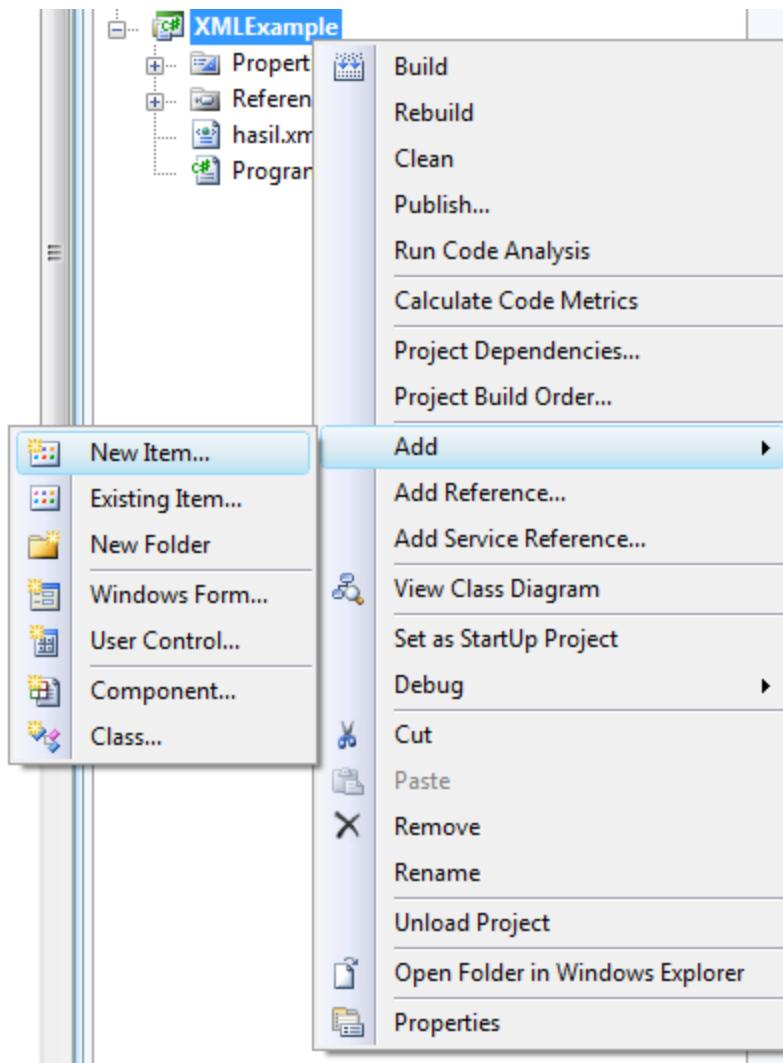
LINQ to XML hadir untuk mengatasi masalah-masalah diatas. Karena dengan adanya LINQ to XML beserta librarynya maka sekarang kita para developer dapat menggunakan Standar Query Operator LINQ untuk melakukan query terhadap dokumen XML. Sebagai perbandingan akan kenyamanan dalam penggunaan LINQ to XML HOL dibawah ini menunjukkan perbedaan penggunaan XPath dan LINQ to XML dalam operasi terhadap dokumen XML.

Task 1 : menciptakan project “XMLExample”

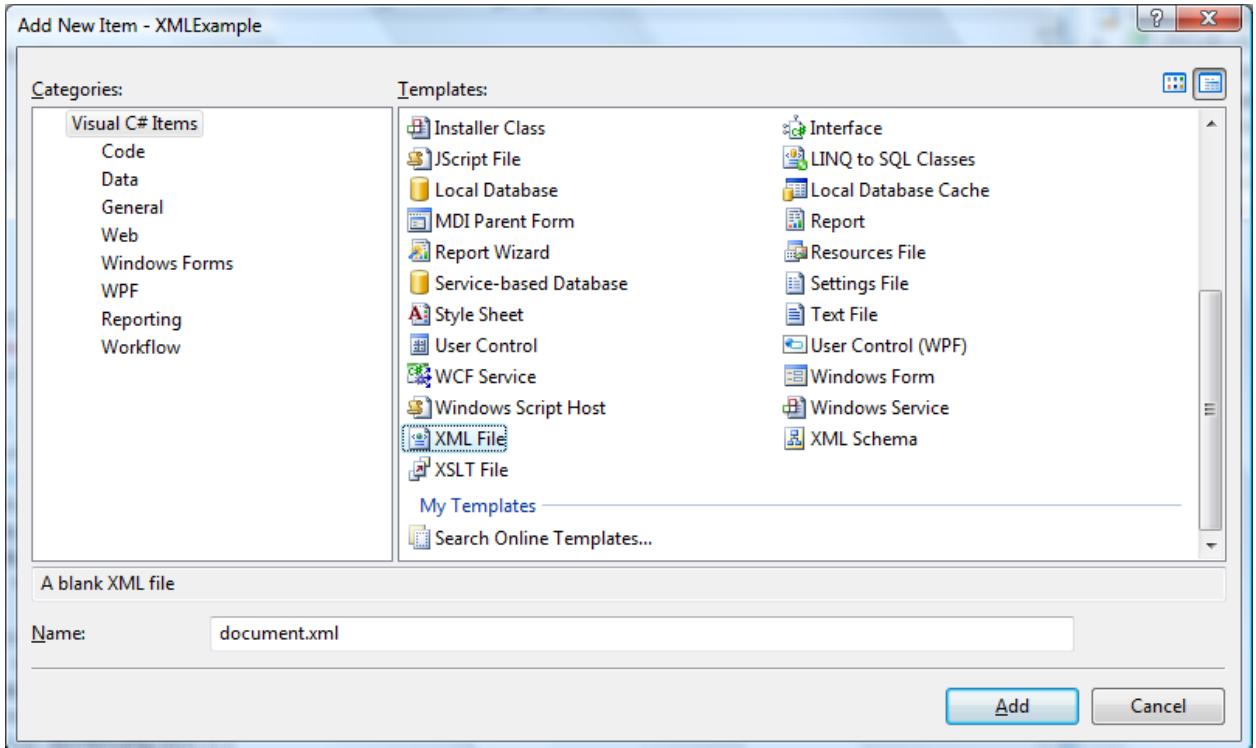
1. Klik Start **Start | Programs | Microsoft Visual Studio 2008 | Microsoft Visual Studio 2008** pada Start Menu.
2. Pada menu **File** pilih **New** dan klik **Project**.
3. Pada kotak dialog **New Project** pilih **Visual C# Windows** .
4. Pilih **Console Application**.
5. Pada kotak nama projek isikan “XMLExample”.
6. Klik **OK**.

Task 2 : membuat file XML

1. Klik kanan pada nama project di “Solution Explorer”



2. Pilih **Add** kemudian **New Item ...**
3. Kemudian pilih tipe **XML File** lalu ketikkan nama “document.xml”



4. Kemudian masukkan data dibawah ini pada file document.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<KumpulanMahasiswa>
    <Mahasiswa Nama="Udin">
        <NIM>13504008</NIM>
        <IPK>3.1</IPK>
    </Mahasiswa>
    <Mahasiswa Nama="Joko">
        <NIM>13504012</NIM>
        <IPK>3.5</IPK>
    </Mahasiswa>
    <Mahasiswa Nama="Saseno">
        <NIM>13504028</NIM>
        <IPK>3.8</IPK>
    </Mahasiswa>
</KumpulanMahasiswa>
```

Task 3 : menuliskan kode untuk operasi terhadap file XML

1. Kemudian buka Program.cs atau Program.vb dan tuliskan kode dibawah ini pada prosedur Main

```
class Program
{
    //path dari file document.xml yang ada di komputer anda
    static string filePath = @"C:\Users\Ronald
Rajagukguk\Documents\Visual
2008\Projects\LINQeBook\XMLExample\document.xml";
    static void Main(string[] args)
    {
        XmlDocument doc = new XmlDocument();
```

```

        //proses pembacaan file document.xml
        doc.Load(filePath);
        //pembuatan objek XPathNavigator yang berguna untuk melakukan
//navigasi dan editing terhadap sebuah dokumen XML
        XPathNavigator navigator = doc.CreateNavigator();
        //menghitung jumlah elemen Mahasiswa yang ada dalam dokumen
        int jumlahElemen = navigator.Select("//Mahasiswa").Count;
        //melakukan navigasi ke elemen Kumpulan Mahasiswa
        navigator.MoveToChild("KumpulanMahasiswa", "");
        //melakukan navigasi ke elemen Mahasiswa yang pertama ditemui
        navigator.MoveToChild("Mahasiswa", "");
        //melakukan loop terhadap isi dari setiap elemen mahasiswa
        // yang ada di dokumen
        for (int i = 0; i < jumlahElemen; i++)
        {
            //navigasi ke elemen IPK yang ada dalam elemen Mahasiswa
            navigator.MoveToChild("IPK", "");
            //mengecek nilai dari elemen IPK
            double tambahan = navigator.ValueAsDouble;
            if (tambahan > 3.5)
                //jika IPK lebih dari 3.5 maka tambahkan sebuah
                //atribut pada elemen IPK
                navigator.CreateAttribute("", "Status", "", "CumLaude");
            //Pindah ke elemen Mahasiswa yang menjadi parent
            navigator.MoveToParent();
            //Pindah ke elemen Mahasiswa selanjutnya
            navigator.MoveToNext();
        }
        //pindah ke root dari dokumen XML
        navigator.MoveToRoot();
        //Menyimpan file ini
        doc.Save(filePath);
    }
}

```

2. Tambahkan penggunaan namespace dibawah ini dibagian atas file **Program.cs**

```

using System.Xml;
using System.Xml.XPath;

```

3. Tekan F5 untuk menjalankan program anda.
4. Setelah program selesai dijalankan periksa file XML anda pastikan bahwa isinya sekarang seperti dibawah ini :

```

<?xml version="1.0" encoding="utf-8"?>
<KumpulanMahasiswa>
    <Mahasiswa Nama="Udin">
        <NIM>13504008</NIM>
        <IPK>3.1</IPK>
    </Mahasiswa>
    <Mahasiswa Nama="Joko">
        <NIM>13504012</NIM>
        <IPK>3.5</IPK>
    
```

```

</Mahasiswa>
<Mahasiswa Nama="Saseno">
    <NIM>13504028</NIM>
    <IPK Status="CumLaude">3.8</IPK>
</Mahasiswa>
</KumpulanMahasiswa>

```

Tentunya hal diatas relatif agak repot karena kita harus melakukan transversal terhadap semua elemen secara manual terutama bagi developer yang terbiasa menggunakan query SQL. Namun contoh dibawah ini akan menunjukkan bagaimana penggunaan LINQ to SQL untuk melakukan hal yang sama dengan diatas. Gantikan task ke 3 pada contoh diatas dengan task ke 3 dibawah ini:

- Kemudian buka Program.cs dan tuliskan kode dibawah ini pada prosedur Main

```

class ProgramEditPakeLINQ
{
    //path dari file document.xml yang ada di komputer anda
    static string FilePath = @"C:\Users\Ronald
Rajagukguk\Documents\Visual
Studio
2008\Projects\LINQeBook\XMLExample\hasil.xml";
    static void Main()
    {
        //pembuatan XDocument yang mewakili sebuah dokumen XML
        XDocument doc = XDocument.Load(FilePath);
        //query untuk mengambil sebuah elemen Mahasiswa dalam dokumen
        XML
        var hasil = from el in doc.Descendants("Mahasiswa")
                    select el;
        //iterasi setiap elemen Mahasiswa
        foreach (var elemen in hasil)
        {
            //mengecek nilai dari elemen IPK
            if (Convert.ToDouble(elemen.Element("IPK").Value) > 3.5)
                //jika IPK lebih dari 3.5 maka tambahkan sebuah
                attribut pada elemen IPK
                elemen.Element("IPK").Add(new
                    XAttribute("Status",
                    "CumLaude"));
            }
        //Menyimpan file ini
        doc.Save(FilePath);
    }
}

```

- Tambahkan penggunaan namespace dibawah ini dibagian atas file **Program.cs**

```
using System.Xml.Linq;
```

- Tekan F5 untuk menjalankan program anda (pastikan bahwa kode anda dicompile dengan dukungan .NET framework 3.5).

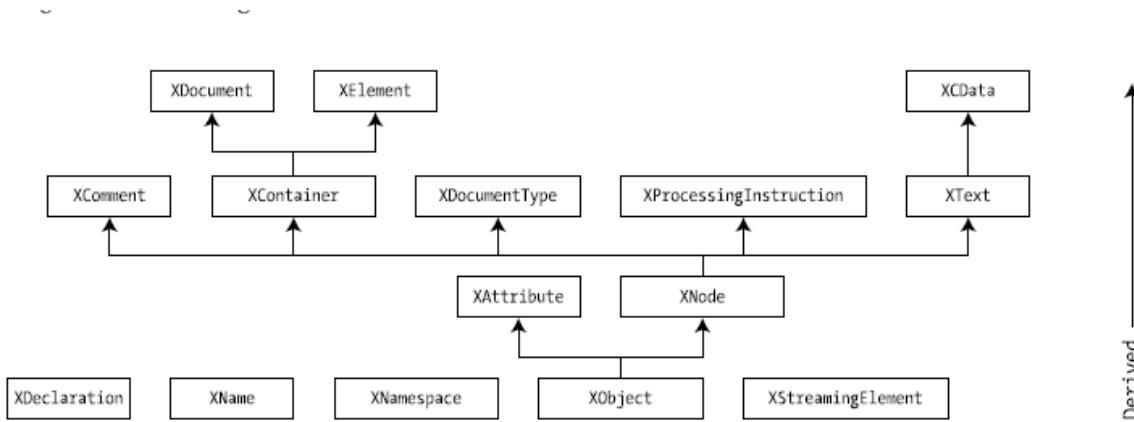
4. Setelah program selesai dijalankan periksa file XML anda pastikan bahwa isinya sekarang seperti dibawah ini :

```
<?xml version="1.0" encoding="utf-8"?>
<KumpulanMahasiswa>
  <Mahasiswa Nama="Udin">
    <NIM>13504008</NIM>
    <IPK>3.1</IPK>
  </Mahasiswa>
  <Mahasiswa Nama="Joko">
    <NIM>13504012</NIM>
    <IPK>3.5</IPK>
  </Mahasiswa>
  <Mahasiswa Nama="Saseno">
    <NIM>13504028</NIM>
    <IPK Status="CumLaude">3.8</IPK>
  </Mahasiswa>
</KumpulanMahasiswa>
```

Dari dua contoh diatas terlihat jelas perbedaan dalam pemrosesan dokumen XML antara LINQ to XML dengan DOM framework yang ada di .NET 2.0, dengan penggunaan LINQ to XML kita memperoleh citarasa deklaratif seperti pada query SQL ketika melakukan query. Sebenarnya bentuk querynya sama dengan ketika kita melakukan query terhadap objek karena memang kita melakukan query terhadap objek yaitu objek XDocument dan XElement yang mewakili dokumen dan elemen dari sebuah data XML. Pembahasan tentang kelas-kelas yang mendukung operasi LINQ to XML akan dibahas pada bab berikutnya

LINQ to XML API

Jika pada bagian sebelumnya anda telah melihat gambaran singkat tentang penggunaan LINQ to XML. Maka mari sekarang kita melihat bagaimana dukungan API untuk XML. API ini bersifat independen terhadap LINQ to XML, atau dengan kata lain developer dapat menggunakan API ini untuk melakukan operasi terhadap dokumen XML tanpa perlu mengoperasikannya dengan LINQ to XML.



Gambar diatas menjelaskan tentang struktur kelas dari API LINQ to XML. Jika melihat diagram diatas maka ada beberapa hal yang menarik yaitu :

1. Beberapa kelas diatas bersifat abstrak, sehingga kita tidak akan bisa membuat objek dari kelas tersebut. Kelas yang abstrak ini adalah Xobject,Xcontainer dan juga Xnode
2. Kelas yang mewakili atribut tidak diturunkan dari sebuah node (XNode) karena sebenarnya atribut bukan merupakan sebuah node.
3. Elemen Streaming (XStreamingElement) tidak memiliki hubungan inheritance dengan elemen (XElement)
4. Kelas XDocument dan XElement adalah dua buah kelas yang diturunkan dari Xnode.

Setelah kita melihat struktur kelas API LINQ to SQL, mari sekarang kita lihat sama-sama bagaimana API tersebut digunakan oleh LINQ untuk berurusan dengan XML.

Pembuatan XML

Seperti telah sama-sama kita lihat pada bagian sebelumnya bahwa dengan adanya API LINQ to SQL pembuatan sebuah dokumen XML akan menjadi relatif lebih mudah. Maka pada bagian ini kita akan melihat bagaimana kemudahan itu diaplikasikan.

Pembuatan Element pada XML dengan menggunakan XElement

Perlu diingat bahwa XElement adalah sebuah kelas yang akan banyak membantu kita dalam perjalanan kita menggunakan LINQ to XML. Konstruktor dari XElement memiliki 5 buah prototype namun pada buku ini saya hanya akan membahas 3 prototype saja yaitu :

```
public XElement(XName name);
public XElement(XName name, object content);
public XElement(XName name, params object[] content);
```

konstruktor yang pertama akan membuat sebuah element dengan nama sesuai dengan nama yang didefinisikan pada parameternya. Sehingga dengan menggunakan konstruktor ini maka akan elemen ini belum memiliki sebuah isi. Contoh penggunaannya adalah :

```
XElement awal = new XElement("Pegawai");
Console.WriteLine(awal);
```

Hasil dari kode diatas adalah :

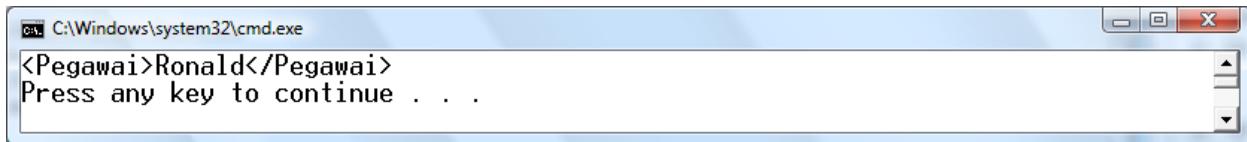


The screenshot shows a standard Windows command prompt window titled 'C:\Windows\system32\cmd.exe'. The window contains the following text:
<Pegawai />
Press any key to continue . . .

Sama seperti konstruktor tipe pertama konstruktor yang kedua pun akan membuat sebuah element dengan nama sesuai dengan nama yang didefinisikan pada parameternya. Namun tipe yang kedua ini memiliki tambahan untuk mendefinisikan isi dari elemen tersebut pada saat pembuatan elemen.. Contoh penggunaan prototype ini adalah sebagai berikut :

```
XElement awal = new XElement("Pegawai","Ronald");
Console.WriteLine(awal);
```

Terlihat bahwa kode diatas akan membuat sebuah elemen dengan nama “Pegawai” dan isinya adalah sebuah String dengan nilai “Ronald”. Hasil dari kode diatas adalah:

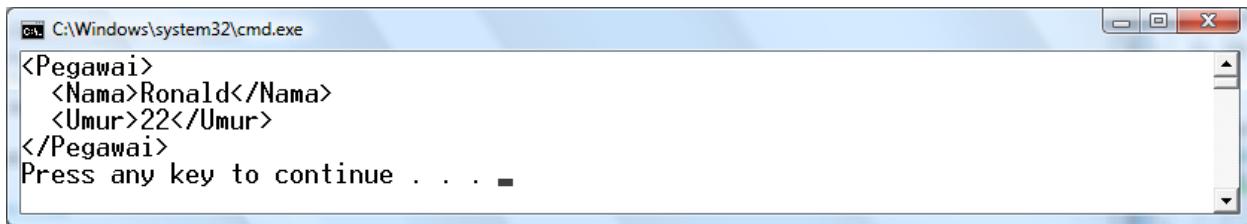


```
C:\Windows\system32\cmd.exe
<Pegawai>Ronald</Pegawai>
Press any key to continue . . .
```

Prototype terakhir yang dibahas mirip seperti prototype yang kedua. Namun perbedaannya adalah kita dapat memasukkan beberapa objek sekaligus kedalam sebuah elemen. Contoh penggunaanya adalah sebagai berikut :

```
XElement awal = new XElement("Pegawai",
    new XElement("Nama", "Ronald"),
    new XElement("Umur", "22")
);
Console.WriteLine(awal);
```

Kode diatas akan menambahkan 2 elemen (elemen dengan nama “Nama” dan “Umur”) kedalam sebuah elemen dengan nama “Pegawai”. Hasil dari kode diatas adalah :



```
C:\Windows\system32\cmd.exe
<Pegawai>
  <Nama>Ronald</Nama>
  <Umur>22</Umur>
</Pegawai>
Press any key to continue . . . -
```

Terlihat bahwa penggunaan prototype yang ketiga akan memudahkan untuk pembuatan sebuah elemen yang mempunyai lebih dari 1 buah elemen anak.

Pembuatan Atribut pada XML dengan menggunakan XAttribute

Atribut adalah sebuah pasangan nama dan nilai yang “ditempelkan” pada sebuah elemen. Dalam API LINQ to XML maka atribut ini diwakili dengan kelas XAttribute yang dapat dimasukkan ke sebuah koleksi dalam objek XElement. Prototype dari konstruktor kelas ini ada 2 buah yaitu :

```
public XAttribute(XAttribute other);

public XAttribute(XName name, object value);
```

Prototype yang pertama tidak lebih dari sebuah konstruktor yang akan menerima masukan kelas XAttribute yang akan dikopi. Prototype yang kedua akan menerima masukan dua buah parameter yaitu nama dan juga objek yang akan menjadi nilai dari atribut tersebut. Aturan yang sama dengan isi dari XElement berlaku juga untuk objek yang menjadi nilai atribut ini. Contoh kode penggunaanya adalah :

```
XElement awal = new XElement("Pegawai");
XAttribute atribut = new XAttribute("Nama", "Ronald");
awal.Add(atribut);
Console.WriteLine(awal);
```

Hasil dari kode diatas dapat dilihat pada gambar dibawah ini. Terlihat bahwa pada elemen dengan nama “Pegawai” ditambahkan sebuah atribut dengan pasangan nama “Nama” dan nilai “Ronald”



```
C:\Windows\system32\cmd.exe
<Pegawai Nama="Ronald" />
Press any key to continue . . .
```

Penulisan Komentar pada XML dengan menggunakan Xcomment

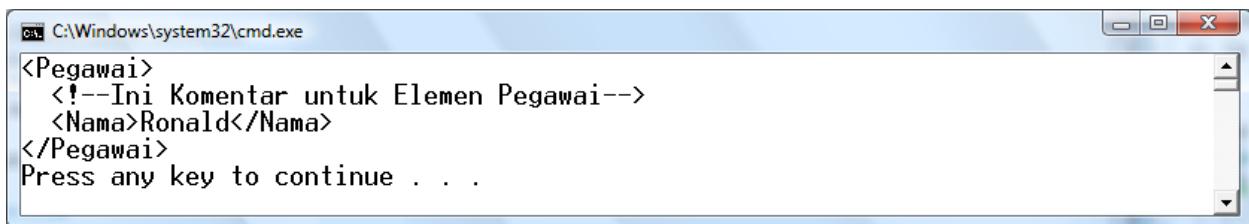
Komentar dalam XML adalah sesuatu hal yang cukup penting, terutama untuk hal yang menyangkut bagaimana pembacaan dokumen XML itu oleh manusia. Pembuatan komentar pada API LINQ to XML didukung oleh kelas Xcomment. Prototype untuk XComment ada dua buah yaitu :

```
public XComment(XComment other);
public XComment(string value);
```

Prototype yang pertama tidak lebih dari sebuah konstruktor yang akan menerima masukan kelas XAttribute yang akan dikopi. Prototype yang kedua akan menerima masukan sebuah string yang akan dituliskan sebagai komentar pada dokumen XML. Contoh penggunaannya adalah sebagai berikut :

```
XElement awal = new XElement("Pegawai");
XComment komentar = new XComment("Ini Komentar untuk Elemen Pegawai");
awal.Add(komentar);
awal.Add(new XElement("Nama", "Ronald"));
Console.WriteLine(awal);
```

Hasil dari kode diatas ditunjukkan pada gambar dibawah ini. Terlihat bahwa string “Ini Komentar untuk Elemen Pegawai” dituliskan dengan diapit oleh “” yang menandakannya sebagai komentar.



```
C:\Windows\system32\cmd.exe
<Pegawai>
  <!--Ini Komentar untuk Elemen Pegawai-->
  <Nama>Ronald</Nama>
</Pegawai>
Press any key to continue . . .
```

Pembuatan Dokumen dengan menggunakan XDocument

XDocument adalah kelas yang mewakili sebuah dokumen XML. Sehingga dengan menggunakan XDocument kita dapat membuat sebuah dokumen XML beserta dengan atribut-atributnya. Prototype untuk kelas ini ada 4 buah yaitu :

```
public XDocument(XDocument other);
public XDocument();
public XDocument(XDeclaration declaration, params object[] content);
```

```
public XDocument(params object[] content);
```

Prototype yang pertama akan menerima masukan sebuah objek XDocument lain yang akan dicopy. Sedangkan parameter kedua akan membuat sebuah objek XDocument kosong . Contoh penggunaan dari prototype yang pertama adalah :

```
XDocument awal = new XDocument();
awal.Add(new XElement("Pegawai"));
awal.Save("namaFile.xml");
```

Jika anda buka file namaFile.xml maka isi dari file tersebut adalah :

```
<?xml version="1.0" encoding="utf-8"?>
<Pegawai />
```

Terlihat bahwa dokumen XML dibuat dengan standar bahwa versinya adalah 1.0 dan encoding “utf-8”. Prototype yang ketiga akan membantu anda untuk menggunakan XDeclaration dalam rangka mengeset nilai atribut dari sebuah dokumen XML. Sedangkan prototype yang keempat akan membantu anda dalam membuat sebuah dokumen XML berikut isinya pada saat konstruksi. Contoh penggunaannya adalah :

```
XDocument awal = new XDocument(new XElement("Pegawai"));
awal.Save("namaFile.xml");
```

Jika anda perhatikan file “namaFile.xml” maka isinya akan sama dengan hasil yang dihasilkan oleh program sebelumnya.

Pembuatan Deklarasi dengan menggunakan Xdeclaration

Deklarasi adalah keterangan dari sebuah dokumen XML dan diwakili oleh kelas XDeclaration. Kelas XDeclaration umumnya akan ditambahkan hanya pada kelas XDocument bukan pada XElement. Meskipun XElement dapat saja menerima masukan tipe ini. Prototype dari XDeclaration ada dua buah yaitu :

```
public XDeclaration(XDeclaration other);

public XDeclaration(string version, string encoding, string standalone);
```

Prototype yang pertama tidak lebih dari sebuah konstruktor yang akan menerima masukan kelas XAttribute yang akan dikopi. Prototype yang kedua akan menerima masukan yang berupa versi dari dokumen XML tersebut (normalnya diisi dengan “1.0”) kemudian tipe encoding yang digunakan dan terakhir sebuah boolean yang menyatakan apakah dokumen ini memerlukan dokumen lain untuk proses pembacaannya.

Contoh dari penggunaan XDeclaration adalah :

```
XDocument awal = new XDocument();
awal.Declaration = new XDeclaration("1.0", "UTF-8", "yes");
awal.Add(new XElement("Pegawai"));
awal.Save("file.xml");
```

Jika anda membuka "file.xml" maka anda akan mendapatkan hasil seperti ini :

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<Pegawai />
```

Terlihat bahwa pada bagian atas file tersebut didefinisikan versi dari dokumen XML beserta dengan tipe encodingnya dan atribut standalone yang menyatakan apakah dokumen tersebut membutuhkan dokumen lain atau tidak.

Penulisan XML

Bagian ini akan khusus membahas bagaimana cara penyimpanan XML. Karena tentunya sebuah dokumen XML setelah mengalami berbagai macam operasi akan disimpan.

Penyimpanan dokumen dengan menggunakan XDocument.Save()

Anda dapat menyimpan dokumen XML anda dengan menggunakan prosedur Save yang ada pada kelas XDocument. Prototype dari prosedur ini ada 5 buah yaitu :

```
public void Save(string fileName);
public void Save(TextWriter textWriter);
public void Save(XmlWriter writer);
public void Save(string fileName, SaveOptions options);
public void Save(TextWriter textWriter, SaveOptions options);
```

Prototype yang pertama akan menyimpan dokumen XML tersebut pada sebuah file dengan nama yang didefinisikan pada string masukan. Contoh penggunaannya adalah :

```
XDocument awal = new XDocument(new XElement("Pegawai"));
awal.Save(@"D:\File.xml");
```

Kode diatas akan menyimpan dokumen XML pada file yang bernama "File.xml" yang berada di drive D pada hardisk anda. Kemudian untuk Prototype yang kedua dokumen XML akan disimpan pada sebuah objek turunan dari TextWriter. Biasanya turunannya berupa stream. Contoh penggunaanya adalah pada kode dibawah ini.

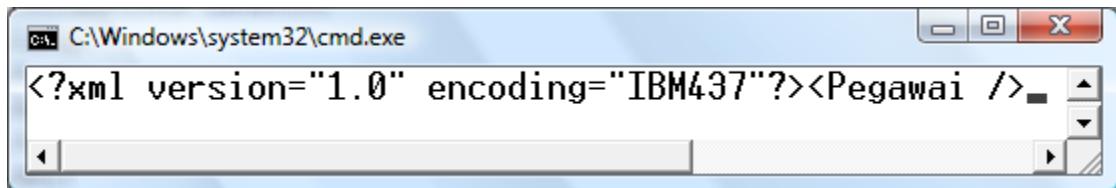
```
XDocument awal = new XDocument(new XElement("Pegawai"));
TextWriter penulis = new StreamWriter(@"D:\File.xml");
awal.Save(penulis);
```

kode diatas akan menyimpan dokumen XML ke sebuah file. Sama persis seperti pada contoh prototype yang pertama. Namun anda pun dapat melakukan eksperimen dengan menuliskan dokumen ke objek stream lainnya. Misalnya Http Writer atau HTMLTextWriter.

Prototype yang ketiga akan melakukan penulisan terhadap sebuah objek turunan dari kelas XmlWriter. Contoh penggunaan dari prototype ini adalah :

```
XDocument awal = new XDocument(new XElement("Pegawai"));
XmlWriter penulis = new XmlTextWriter(Console.Out);
awal.Save(penulis);
```

Karena objek dari kelas turunan XmlWriter yang digunakan adalah XmlTextWriter dengan masukan sebuah objek stream yang akan menuliskan hasil di layar (Console.Out) maka dokumen akan dituliskan langsung dilayar.



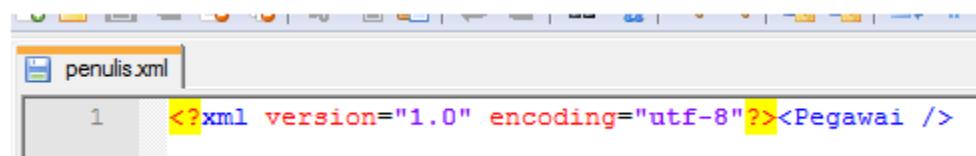
Jika anda ingin menggunakan prototype ketiga ini untuk menuliskan dokumen ke file anda cukup mengganti baris pembuatan objek XmlWriter menjadi seperti dibawah ini :

```
XmlWriter penulis = new XmlTextWriter(@"D:\File.xml", Encoding.Default);
```

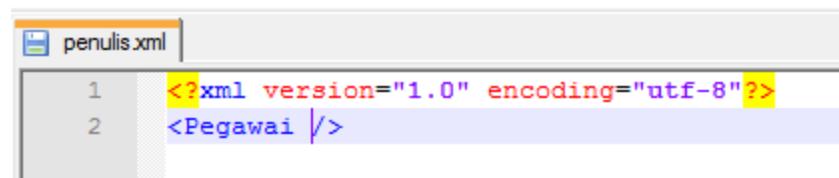
Prototype yang keempat mirip seperti yang pertama, Perbedaanya adalah pada adanya tambahan sebuah parameter yaitu parameter berupa enum dengan tipe SaveOptions. Enum SaveOptions ini memiliki 2 opsi yaitu DisableFormatting dan None. Jika kita memilih DisableFormatting maka hasil keluaran dari dokumen XML kita tidak akan memiliki indentasi. Contoh penggunaan prototype ini dengan opsi DisableFormatting adalah :

```
XDocument awal = new XDocument(new XElement("Pegawai"));
awal.Save(@"D:\namaFile.xml", SaveOptions.DisableFormatting);
```

Maka jika kita membaca file namaFile.xml maka isi dari file tersebut adalah :



Terlihat bahwa elemen pegawai akan langsung dituliskan disamping elemen xml. Jika kita menggunakan opsi None maka dokumen akan tampil seperti dibawah ini :



Prototype yang kelima juga memiliki perilaku yang sama dengan prototype ke empat. Sehingga indentasi dapat diatur akan dihilangkan atau tidak saat anda menuliskan dokumen XML ke sebuah objek TextWriter.

Penyimpanan elemen XML dengan menggunakan XElement.Save()

Pada API LINQ to XML kita tidak perlu harus selalu menggunakan XDocument untuk membuat sebuah dokumen XML. Melainkan cukup dengan menggunakan XElement kita pun sudah dapat membuat sebuah dokumen XML, meskipun kita tidak dapat menambahkan XDeclaration pada XElement. Sama seperti XDocument maka XElement memiliki 5 buah prototype yaitu :

```
public void Save(string fileName);  
  
public void Save(TextWriter textWriter);  
  
public void Save(XmlWriter writer);  
  
public void Save(string fileName, SaveOptions options);  
  
public void Save(TextWriter textWriter, SaveOptions options);
```

Prototype yang pertama akan menyimpan dokumen XML tersebut pada sebuah file dengan nama yang didefinisikan pada string masukan. Contoh penggunaannya adalah :

```
XElement pegawai = new XElement("Pegawai",  
    new XElement("Nama", "Ronald"),  
    new XElement("NIM", "13503117")  
);  
  
pegawai.Save(@"D:\penulis.xml");
```

Kode diatas akan menyimpan dokumen XML pada file yang bernama “File.xml” yang berada di drive D pada hardisk anda. Kemudian untuk Prototype yang kedua dokumen XML akan disimpan pada sebuah objek turunan dari TextWriter. Biasanya turunannya berupa stream. Contoh penggunaanya adalah pada kode dibawah ini.

```
XElement pegawai = new XElement("Pegawai",  
    new XElement("Nama", "Ronald"),  
    new XElement("NIM", "13503117")  
);  
TextWriter penulis = new StreamWriter(@"D:/penulis.xml");  
pegawai.Save(penulis);
```

Kode diatas akan menyimpan dokumen XML ke sebuah file. Sama persis seperti pada contoh prototype yang pertama. Namun anda pun dapat melakukan eksperimen dengan menuliskan dokumen ke objek stream lainnya. Misalnya Http Writer atau HTMLTextWriter.

Prototype yang ketiga akan melakukan penulisan terhadap sebuah objek turunan dari kelas XmlWriter. Contoh penggunaan dari prototype ini adalah :

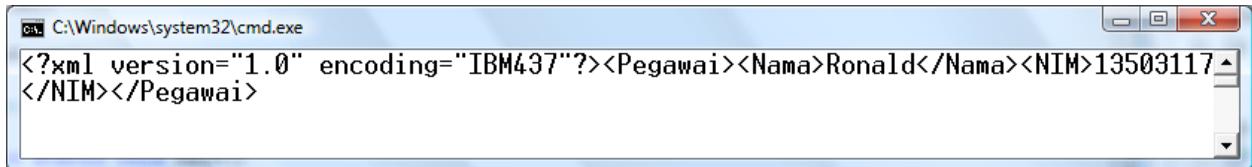
```
XElement pegawai = new XElement("Pegawai",
```

```

        new XElement("Nama", "Ronald"),
        new XElement("NIM", "13503117")
    );
XmlWriter penulis = new XmlTextWriter(Console.Out);
pegawai.Save(penulis);

```

Karena objek dari kelas turunan XmlWriter yang digunakan adalah XmlTextWriter dengan masukan sebuah objek stream yang akan menuliskan hasil di layar (Console.Out) maka dokumen akan dituliskan langsung dilayar.



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The output of the XML code is displayed, showing a single-line XML document:

```
<?xml version="1.0" encoding="IBM437"?><Pegawai><Nama>Ronald</Nama><NIM>13503117</NIM></Pegawai>
```

Jika anda ingin menggunakan prototype ketiga ini untuk menuliskan dokumen ke file anda cukup mengganti baris pembuatan objek XmlWriter menjadi seperti dibawah ini :

```
XmlWriter penulis = new XmlTextWriter(@"D:\File.xml", Encoding.Default);
```

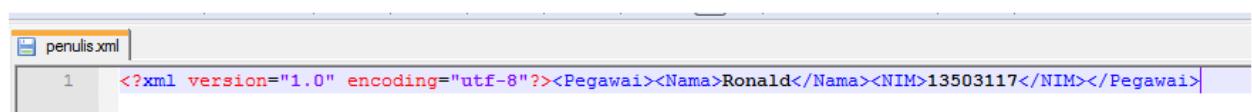
Prototype yang keempat mirip seperti yang pertama, Perbedaanya adalah pada adanya tambahan sebuah parameter yaitu parameter berupa enum dengan tipe SaveOptions. Enum SaveOptions ini memiliki 2 opsi yaitu DisableFormatting dan None. Jika kita memilih DisableFormatting maka hasil keluaran dari dokumen XML kita tidak akan memiliki indentasi. Contoh penggunaan prototype ini dengan opsi DisableFormatting adalah :

```

 XElement pegawai = new XElement("Pegawai",
    new XElement("Nama", "Ronald"),
    new XElement("NIM", "13503117")
);
pegawai.Save(@"D:\namaFile.xml", SaveOptions.DisableFormatting);

```

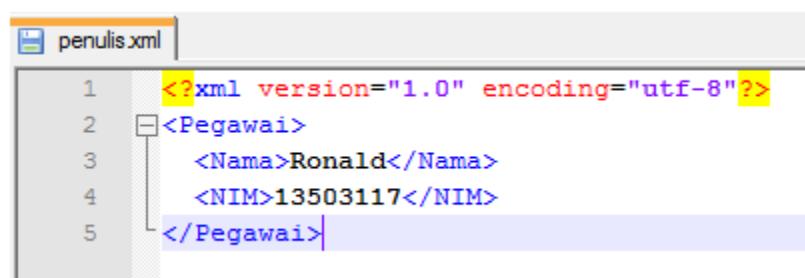
Maka jika kita membaca file namaFile.xml maka isi dari file tersebut adalah :



The screenshot shows a Notepad window titled 'penulis.xml'. The content of the file is a single-line XML document:

```
<?xml version="1.0" encoding="utf-8"?><Pegawai><Nama>Ronald</Nama><NIM>13503117</NIM></Pegawai>
```

Terlihat bahwa elemen pegawai akan langsung dituliskan disamping elemen xml. Jika kita menggunakan opsi None maka dokumen akan tampil seperti dibawah ini :



The screenshot shows a Notepad window titled 'penulis.xml'. The content of the file is a multi-line XML document with indentation:

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <Pegawai>
3      <Nama>Ronald</Nama>
4      <NIM>13503117</NIM>
5  </Pegawai>

```

Prototype yang kelima juga memiliki perilaku yang sama dengan prototype ke empat. Sehingga indentasi dapat diatur akan dihilangkan atau tidak saat anda menuliskan dokumen XML ke sebuah objek TextWriter.

Pembacaan XML

Bagian ini akan khusus membahas bagaimana cara membaca sebuah dokumen XML agar dapat menjadi sebuah tree XML.

Membaca Dokumen dengan menggunakan XDocument.Load()

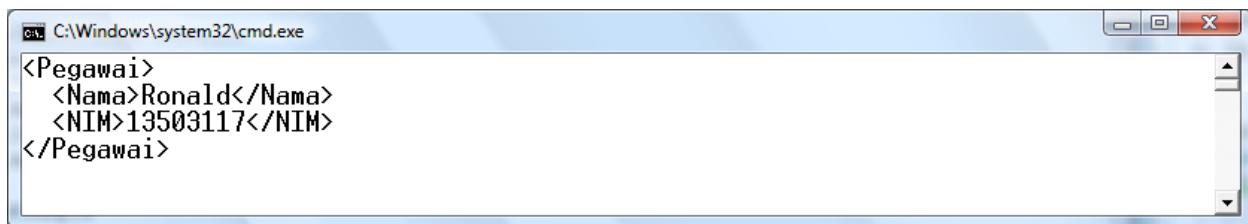
Anda dapat membaca sebuah dokumen dengan menggunakan prosedur Load yang ada pada sebuah kelas XDocument. Prototype dari kelas ini ada 6 buah yaitu :

```
public static XDocument Load(string uri);  
  
public static XDocument Load(TextReader textReader);  
  
public static XDocument Load(XmlReader reader);  
  
public static XDocument Load(string uri, LoadOptions options);  
  
public static XDocument Load(TextReader textReader, LoadOptions options);  
  
public static XDocument Load(XmlReader reader, LoadOptions options);
```

Meskipun ada 6 prototype namun jika anda jeli sebenarnya pada intinya ada 3 prototype utama, 3 prototype lainnya hanyalah pelengkap dengan sebuah opsi tambahan yaitu tipe enum LoadOptions. Prototype yang pertama akan menerima masukan sebuah string yang merepresentasikan lokasi dari sebuah file XML. Contoh penggunaanya adalah :

```
XDocument doc = XDocument.Load(@"D:\namaFile.xml");  
Console.WriteLine(doc);
```

Kode diatas akan meload dokumen XML yang sebelumnya telah kita buat pada bagian penulisan dokumen XML. Hasil dari kode ini adalah :



```
<Pegawai>
  <Nama>Ronald</Nama>
  <NIM>13503117</NIM>
</Pegawai>
```

Prototype kedua akan membaca dokumen XML yang bersumber dari sebuah objek TextReader. Contoh penggunaanya adalah dibawah ini

```
TextReader pembaca = new StreamReader(@"D:\namaFile.xml");
XDocument doc = XDocument.Load(pembaca);
Console.WriteLine(doc);
```

Efek dari pemakaian kode diatas akan sama dengan pemakaian kode sebelumnya. Hal ini karena objek TextReadernya merupakan sebuah StreamReader yang akan membaca sebuah file.

Prototype yang ketiga akan melakukan pembacaan file XML yang ada pada sebuah objek XmlReader. Contoh penggunaanya adalah :

```
XmlReader pembaca = new XmlTextReader(@"D:\namaFile.xml");
XDocument doc = XDocument.Load(pembaca);
Console.WriteLine(doc);
```

Karena melakukan pembacaan dari sebuah objek XmlReader yang menunjuk ke sebuah stream yang akan membaca sebuah file. Hasil dari kode diatas akan sama efeknya seperti pemakaian kode pada prototype yang pertama.

Kemudian Tiga prototype yang terakhir akan menerima sebuah parameter tambahan yang berupa tipe enum LoadOptions. Keterangan dari masing-masing opsi dalam tipe enum ini dapat dilihat dibawah :

Opsi	Keterangan
LoadOptions.None	Opsi ini dipilih jika kita tidak memilih opsi lainnya
LoadOptions.PreserveWhiteSpace	Jika opsi ini dipilih maka sebuah elemen teks yang hanya berisikan spasi akan dianggap sebagai sebuah nilai
LoadOptions.SetLineInfo	Jika opsi ini dipilih maka kita dapat mengetahui posisi baris dari objek kelas Xobject pada dokumen XML. Hal ini dilakukan dengan menggunakan interface IxmlLineInfo
LoadOptions.SetBaseUri	Jika opsi ini dipilih maka kita dapat mengetahui alamat URI dari dokumen yang sedang dibaca

Opsi diatas dapat dikombinasikan dengan menggunakan operator bitwise OR (|), namun perlu diperhatikan pada beberapa kasus ada opsi yang tidak dapat diterapkan. Sebagai contoh ketika kita membuat sebuah dokumen XML dari sebuah string maka tentunya kita tidak akan dapat memperoleh info tentang baris dari dokumen tersebut.

```
<?xml version="1.0" encoding="utf-8"?>
<KumpulanPegawai>
    <Pegawai>
        <Nama>Ronald</Nama>
        <NIM>13503117</NIM>
    </Pegawai>
    <Pegawai>
        <Nama>          </Nama>
        <NIM>13503118</NIM>
    </Pegawai>
</KumpulanPegawai>
```

Contoh kode dibawah ini akan menunjukkan bagaimana penggunaan opsi LoadOptions.PreserveWhitespace untuk dokumen diatas :

```

XDocument doc = XDocument.Load(@"D:\namaFile.xml",
    LoadOptions.PreserveWhitespace);
foreach (var turunan in doc.Descendants("Nama"))
    Console.WriteLine("Panjang isi elemen {0} Karakter"
        , turunan.Value.Length);

```

Maka hasil dari kode diatas adalah :

Terlihat bahwa spasi pada elemen Nama dianggap sebagai nilai dari elemen tersebut, hal ini terbukti dari dihitungnya karakter spasi dalam nilai elemen. Namun jika kita menggunakan opsi LoadOptions.None maka hasilnya adalah sebagai berikut :

Nilai string pada elemen yang kedua akan dihitung sebagai 0 karakter. Hal ini karena jika opsi LoadOptions.PreserveWhitespace tidak dipilih maka karakter whitespace seperti spasi atau enter tidak akan dianggap sebagai data.

Kemudian untuk penggunaan opsi SetLineInfo adalah sebagai berikut :

```

XDocument doc = XDocument.Load(@"D:\namaFile.xml",
    LoadOptions.SetLineInfo);
foreach (var turunan in doc.Descendants("Nama"))
    Console.WriteLine("Nomor baris = {0} posisi :{1}",
        ((IXmlLineInfo)turunan).LineNumber,
        ((IXmlLineInfo)turunan).LinePosition
    );

```

Kode diatas akan menuliskan hasil

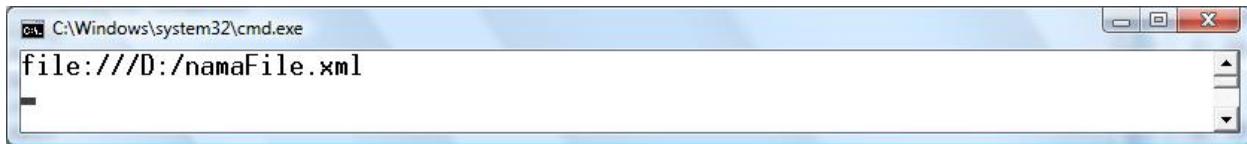
Hasil pada baris pertama menuliskan bahwa nomor baris adalah 4 karena elemen "Nama" yang pertama berada pada baris ke 4 dari dokumen XML dan seterusnya.Selanjutnya penggunaan opsi SetBaseUri adalah sebagai berikut :

```

XDocument doc = XDocument.Load(@"D:\namaFile.xml",
    LoadOptions.SetBaseUri);
Console.WriteLine(doc.BaseUri);

```

Hasil dari kode diatas adalah :



Dilayar tertulis lokasi dari dokumen XML yang sedang kita baca.

Membaca Dokumen dengan menggunakan XElement.Load()

Seperti telah disinggung pada bab sebelumnya bahwa untuk menulis dokumen XML kita dapat juga untuk tidak selalu menggunakan XDocument melainkan dapat menggunakan XElement. Hal yang sama pun berlaku ketika kita akan melakukan pembacaan terhadap dokumen XML. Kita dapat menggunakan XElement. Penggunaannya relatif sama dengan penggunaan XDocument.Load(). Prototype dari prosedur ini pun sama seperti XElement.Load() ada 6 buah yaitu :

```
public static XElement Load(string uri);  
  
public static XElement Load(TextReader textReader);  
  
public static XElement Load(XmlReader reader);  
  
public static XElement Load(string uri, LoadOptions options);  
  
public static XElement Load(TextReader textReader, LoadOptions options);  
  
public static XElement Load(XmlReader reader, LoadOptions options);
```

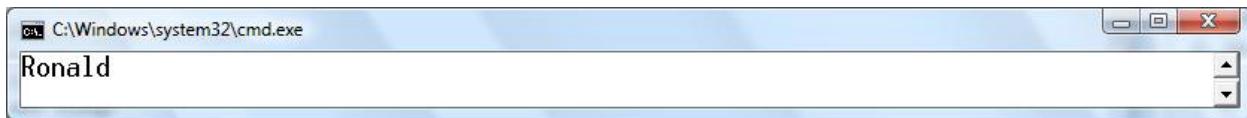
Contoh penggunannya sama dengan ketika kita menggunakan XDocument.Load().

Parsing Dokumen dengan menggunakan XDocument.Parse() atau XElement.Parse()

Dalam pengembangan perangkat lunak sering kali kita memperoleh sebuah dokumen XML dalam bentuk string. Dapat dibayangkan bagaimana repotnya untuk memarsing string tersebut menjadi sebuah XML Tree, namun dengan menggunakan fungsi XDocument.Parse() dan XElement.Parse() yang akan menerima masukan sebuah string dan mengembalikan sebuah objek XDocument ataupun XElement. Contoh kode penggunaanya adalah sebagai berikut :

```
string StringXML =  
"<Pegawai><Nama>Ronald</Nama><NIM>13503117</NIM></Pegawai>";  
XDocument elem = XDocument.Parse(StringXML);  
Console.WriteLine(elem.Element("Pegawai").Element("Nama").Value);
```

Kode diatas akan menuliskan nilai dari elemen "Nama" pertama yang ada dalam elemen "Pegawai". Dengan adanya fungsi ini maka tentunya kita relatif tidak akan dipusingkan lagi dengan urusan parsing string ke bentuk XML. Hasil dari kode diatas adalah :



Penelusuran Dokumen XML

Pada bagian ini akan dibahas mengenai bagaimana kita dapat melakukan penelusuran Dokumen XML dengan menggunakan API LINQ to SQL. Dalam API LINQ to SQL terdapat 4 properti dan 11 prosedur yang dapat kita gunakan untuk melakukan penelusuran dokumen. Kemudian untuk para developer yang telah terbiasa menggunakan XPath maka dapat menggunakan XPath untuk melakukan penelusuran terhadap dokumen XML.

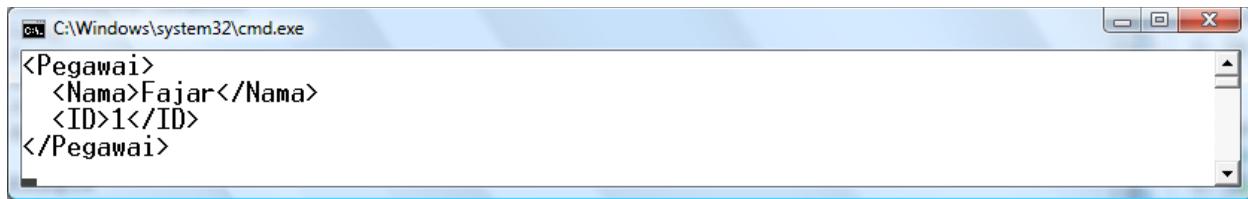
XNode.NextNode

NextNode merupakan sebuah properti pada XNode. Properti ini akan menunjuk kepada node yang ada setelah node yang bersangkutan. Sehingga dengan menggunakan properti ini kita akan mendapatkan efek tranversal “maju”. Contoh penggunaan properti ini adalah sebagai berikut :

```
 XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama = new XElement("Pegawai",
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11"))

    ,
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))
));
Console.WriteLine(PegawaiPertama.NextNode);
```

Terlihat bahwa dari kode diatas objek PegawaiPertama adalah objek XElement yang pertama kali dimasukkan kedalam sebuah XElement dengan nama “KumpulanMahasiswa”. Kemudian setelah dimasukkan objek PegawaiPertama maka dimasukkan lagi sebuah XElement untuk menjadi anak ke dua dari objek XElement “KumpulanMahasiswa”. Sehingga nilai dari properti NextNode dari objek PegawaiPertama akan diisi oleh objek anak ke dua dari objek XElement. Hasil dari kode diatas adalah :



```
C:\Windows\system32\cmd.exe
<Pegawai>
<Nama>Fajar</Nama>
<ID>1</ID>
</Pegawai>
```

XNode.PreviousNode

PreviousNode berlawanan dengan NextNode akan menunjuk ke Node yang berada sebelum node yang bersangkutan. Sehingga dengan menggunakan properti ini kita akan mendapatkan efek tranversal “mundur”. Contoh penggunaan properti ini adalah sebagai berikut :

```
 XElement PegawaiKedua;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    new XElement("Pegawai",
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11"))

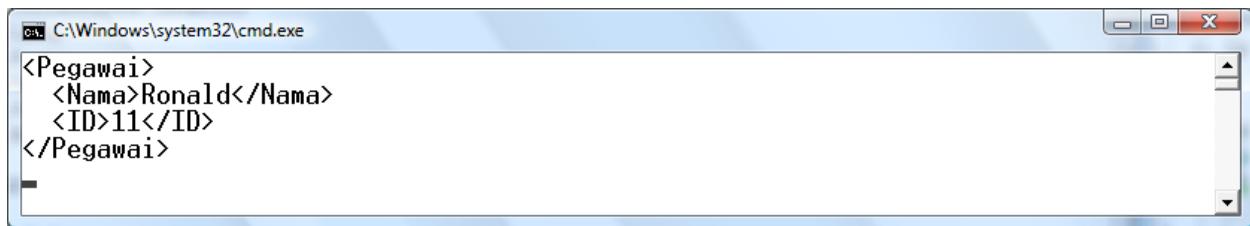
    ,
    PegawaiKedua= new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
```

```

        new XElement("ID", "1"))
    );
Console.WriteLine(PegawaiKedua.PreviousNode);

```

Terlihat bahwa dari kode diatas objek PegawaiPertama adalah objek XElement yang dimasukkan kedua kedalam sebuah XElement dengan nama “KumpulanMahasiswa”. Sebelum pemasukan objek PegawaiPertama telah dimasukkan sebuah XElement untuk menjadi anak pertama dari objek XElement “KumpulanMahasiswa”. Sehingga nilai dari properti PreviousNode dari objek PegawaiKedua akan diisi oleh objek anak pertama dari objek XElement. Hasil dari kode diatas adalah :



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The output displayed is:

```

<Pegawai>
  <Nama>Ronald</Nama>
  <ID>11</ID>
</Pegawai>

```

XObject.Document

Untuk memperoleh sebuah dokumen XML yang menampung sebuah objek XObject yang bersangkutan maka kita dapat menggunakan properti Document dari objek tersebut. Contoh kode penggunaannya adalah :

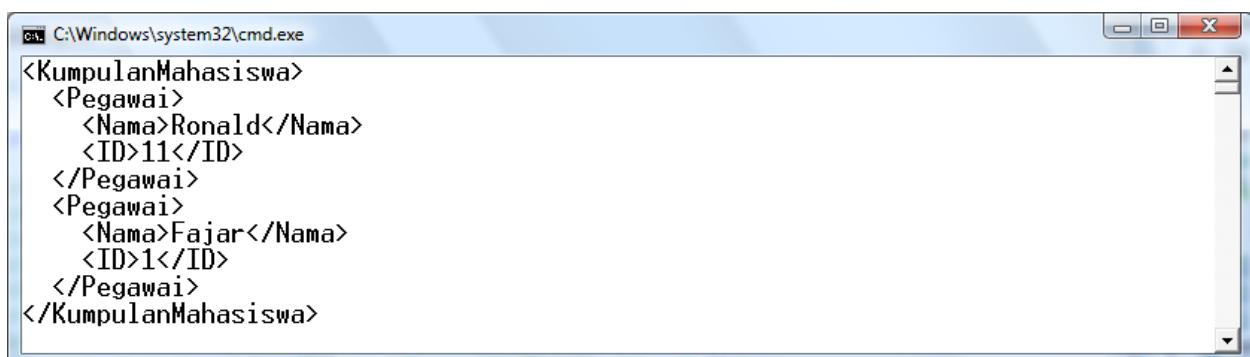
```

 XElement PegawaiPertama;
 XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama = new XElement("Pegawai",
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11"))

    ,
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))
    ));
Console.WriteLine(PegawaiPertama.Document);

```

Hasil dari kode diatas adalah :



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The output displayed is:

```

<KumpulanMahasiswa>
  <Pegawai>
    <Nama>Ronald</Nama>
    <ID>11</ID>
  </Pegawai>
  <Pegawai>
    <Nama>Fajar</Nama>
    <ID>1</ID>
  </Pegawai>
</KumpulanMahasiswa>

```

Terlihat bahwa pada layar akan dituliskan sebuah dokumen XML yang mengandung objek PegawaiPertama.

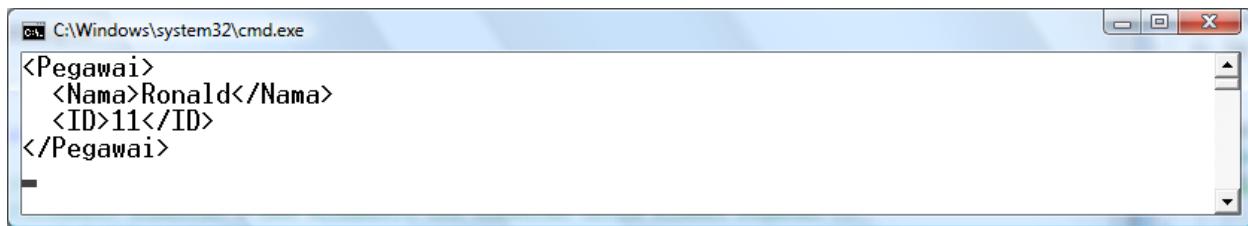
XObject.Parent

Jika anda ingin melakukan transversal 1 elemen ke atas dalam sebuah dokumen XML maka anda dapat menggunakan properti Parent yang ada dalam XObject. Contoh penggunaanya adalah :

```
XElement NamaPegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    new XElement("Pegawai",
        NamaPegawaiPertama= new XElement("Nama", "Ronald"),
        new XElement("ID", "11"))

    ,
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))
));
Console.WriteLine(NamaPegawaiPertama.Parent);
```

Terlihat pada kode diatas bahwa objek XElement NamaPegawaiPertama merupakan anak dari sebuah XElement yang bernama “Pegawai” sehingga ketika properti parent dari objek NamaPegawaiPertama dipanggil maka dilayar akan dituliskan hasil seperti dibawah ini.



```
C:\Windows\system32\cmd.exe
<Pegawai>
<Nama>Ronald</Nama>
<ID>11</ID>
</Pegawai>
```

Prosedur untuk melakukan transversal dalam Dokumen XML

Seperti telah disebutkan sebelumnya bahwa ada 11 prosedur untuk melakukan proses transversal dalam dokumen XML.

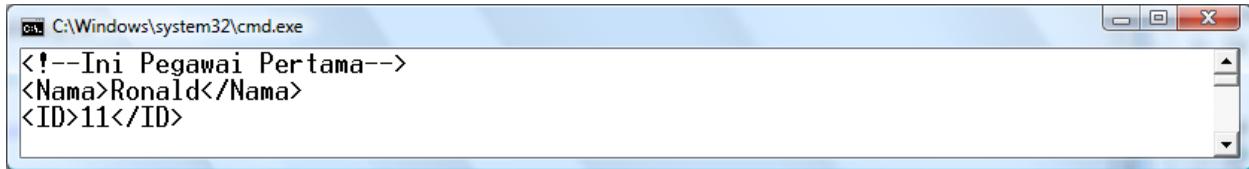
XContainer.Nodes()

Prosedur Nodes() dalam kelas XContainer akan mengembalikan semua objek anak yang berupa objek XNode. Objek kembalian dari prosedur ini akan berupa sebuah koleksi IEnumerable<T>, sehingga kita pun dapat menggunakan SQO terhadap nilai kembalian dari prosedur ini. Contoh penggunaan prosedur ini adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama= new XElement("Pegawai",
        new XComment("Ini Pegawai Pertama"),
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11"))

    ,
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))
));
foreach (var hasil in PegawaiPertama.Nodes())
    Console.WriteLine(hasil);
```

Terlihat dalam kode diatas bahwa kita berusaha untuk menampilkan setiap elemen yang ada dalam objek PegawaiPertama. Sehingga hasil dari kode diatas adalah :



```
C:\Windows\system32\cmd.exe
<!--Ini Pegawai Pertama-->
<Nama>Ronald</Nama>
<ID>11</ID>
```

Jika anda perhatikan maka objek XComment pun akan ikut dituliskan di layar. Hal ini karena XComment adalah kelas turunan dari XNode. Jika anda hanya ingin mendapatkan objek yang berupa XElement maka ubah kode diatas menjadi seperti dibawah ini :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama= new XElement("Pegawai",
        new XComment("Ini Pegawai Pertama"),
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11"))

    ,
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))
));
foreach ( XElement hasil in
    PegawaiPertama.Nodes() .OfType< XElement>())
    Console.WriteLine(hasil.Value);
```

Terlihat bahwa kita menggunakan Standard Query Operator (SQO) yang berupa OfType< T > pada prosedur Nodes(), sehingga kita dapat memfilter tipe elemen anak apa saja yang kita ingin ambil dari koleksi. Dalam kode diatas kita memfilter untuk hanya mengambil elemen anak yang merupakan objek XElement sehingga hasil dari kode diatas adalah sebagai berikut :



```
C:\Windows\system32\cmd.exe
<Nama>Ronald</Nama>
<ID>11</ID>
```

XContainer.Elements()

Fungsionalitas dari prosedur ini hampir sama dengan prosedur Nodes(), hanya saja perbedaannya pada prosedur ini hanya objek dengan tipe Xelements saja yang dimasukkan pada koleksi IEnumerable< T > yang menjadi kembalian. Hal ini mirip dengan jika kita menambahkan SQO OfType< T > pada hasil dari kembalian Nodes(). Contoh penggunaannya adalah sebagai berikut :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama= new XElement("Pegawai",
        new XComment("Ini Pegawai Pertama"),
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11"))

    ,
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))));
```

```

        new XElement("Pegawai",
            new XElement("Nama", "Fajar"),
            new XElement("ID", "1"))
    );
    foreach ( XElement hasil in PegawaiPertama.Elements())
        Console.WriteLine(hasil);

```

Sama seperti contoh kode terakhir pada bagian sebelumnya, maka dilayar akan dituliskan objek yang bertipe XElement saja. Hasil dari kode diatas adalah :

```
C:\Windows\system32\cmd.exe
<Nama>Ronald</Nama>
<ID>11</ID>
```

XContainer.Element()

Prosedur Element() ini hanya akan mengembalikan elemen anak yang pertama dari sebuah objek XContainer. Sehingga berbeda dengan 2 prosedur sebelumnya prosedur ini tidak akan mengembalikan sebuah koleksi. Elemen anak yang dikembalikan adalah elemen anak yang memiliki nama yang sama dengan yang dispesifikasi pada string yang menjadi parameter masukan. Contoh kode penggunaan prosedur ini adalah :

```

 XElement PegawaiPertama;
 XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama= new XElement("Pegawai",
        new XComment("Ini Pegawai Pertama"),
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11"))

    ,
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))
));
Console.WriteLine(PegawaiPertama.Element("Nama"));

```

Hasil dari kode diatas adalah :

```
C:\Windows\system32\cmd.exe
<Nama>Ronald</Nama>
```

XNode.Ancestor()

Prosedur ini akan mengembalikan semua elemen yang menjadi parent dari sebuah objek XNode. Fungsi yang diberikannya mirip seperti pada properti Parent. Namun jika pada properti parent hanya akan memberikan elemen yang langsung menjadi parent. Sedangkan pada prosedur Ancestor akan dilakukan sebuah fungsi rekursif yang mengembalikan semua elemen yang menjadi parent dari objek tersebut baik langsung maupun tidak langsung. Contoh kode penggunaan prosedur ini adalah :

```

 XElement NamaPegawaiPertama;
 XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    new XElement("Pegawai",
        new XComment("Ini Pegawai Pertama"),

```

```

NamaPegawaiPertama= new XElement("Nama", "Ronald"),
new XElement("ID", "11"))

        new XElement("Pegawai",
            new XElement("Nama", "Fajar"),
            new XElement("ID", "1"))
    );
foreach ( XElement hasil in NamaPegawaiPertama.Ancestors())
    Console.WriteLine(hasil.Name);

```

Hasil dari kode diatas adalah :

```
C:\Windows\system32\cmd.exe
Pegawai
KumpulanPegawai
```

Terlihat bahwa semua elemen yang menjadi parent baik secara langsung maupun tidak langsung dari objek NamaPegawaiPertama akan ditampilkan dilayar.

XElement.AncestorAndSelf()

Fungsi yang ditawarkan oleh prosedur ini mirip dengan prosedur Ancestor(), perbedaanya adalah dalam fungsi ini elemen yang terkait akan dimasukkan ke dalam koleksi yang menjadi kembalian dari prosedur ini. Contoh kode pengunaan prosedur ini adalah :

```

XElement NamaPegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    new XElement("Pegawai",
        new XComment("Ini Pegawai Pertama"),
        NamaPegawaiPertama= new XElement("Nama", "Ronald"),
        new XElement("ID", "11"))

        new XElement("Pegawai",
            new XElement("Nama", "Fajar"),
            new XElement("ID", "1"))
    );
foreach ( XElement hasil in NamaPegawaiPertama.AncestorsAndSelf())
    Console.WriteLine(hasil.Name);

```

Hasil dari kode diatas adalah :

```
C:\Windows\system32\cmd.exe
Nama
Pegawai
KumpulanPegawai
```

Jika anda perhatikan pada hasil diatas maka dilayar akan dituliskan nama dari elemen yang terkait (objek NamaPegawaiPertama)

XContainer.Descendants()

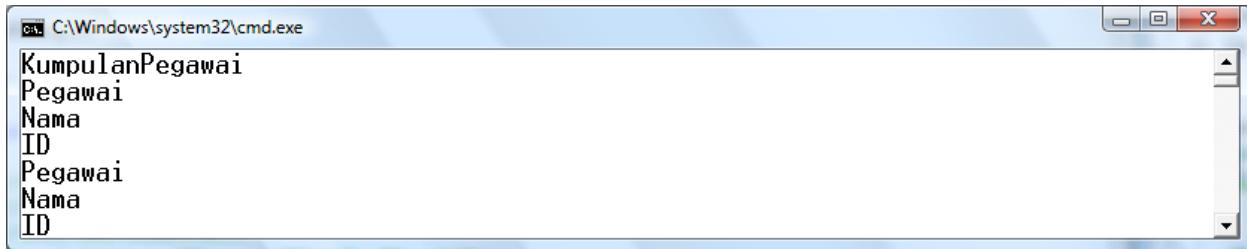
Prosedur ini akan mengembalikan semua elemen yang menjadi anak dari objek XContainer terkait. Elemen yang dikembalikan tidak hanya elemen yang menjadi anak secara langsung melainkan juga yang

menjadi turunan secara tidak langsung. Prosedur ini akan melakukan rekursif untuk menemukan semua elemen yang menjadi turunan. Contoh kode penggunaan prosedur ini adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama = new XElement("Pegawai",
        new XComment("Ini Pegawai Pertama"),
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11"))

    ,
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))
)
);
foreach ( XElement hasil in doc.Descendants())
    Console.WriteLine(hasil.Name);
```

Hasil dari kode diatas adalah :



```
C:\Windows\system32\cmd.exe
KumpulanPegawai
Pegawai
Nama
ID
Pegawai
Nama
ID
```

Terlihat dalam hasil dari kode diatas bahwa seluruh elemen yang menjadi turunan dari objek XDocument ditampilkan namanya.

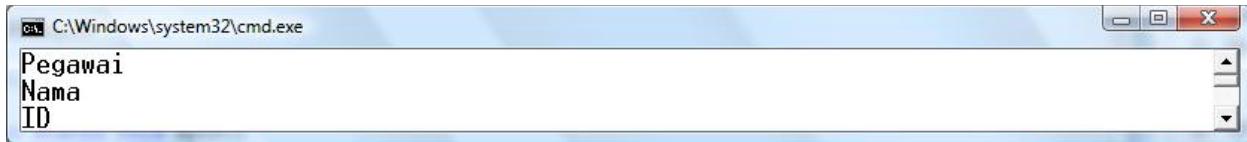
XElement.DescendatsAndSelf()

Fungsi yang ditawarkan oleh prosedur ini mirip dengan prosedur Descendants(), perbedaanya adalah dalam fungsi ini elemen yang terkait akan dimasukkan ke dalam koleksi yang menjadi kembalian dari prosedur ini. Kemudian jika anda jeli maka tentu dapat melihat bahwa fungsi ini dimiliki oleh XElement bukan XDocument sehingga contoh kode pada bagian Descendants() tidak dapat diterapkan sama persis. Contoh kode penggunaan prosedur ini adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama = new XElement("Pegawai",
        new XComment("Ini Pegawai Pertama"),
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11"))

    ,
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))
)
);
foreach ( XElement hasil in PegawaiPertama.DescendatsAndSelf())
    Console.WriteLine(hasil.Name);
```

Hasil dari kode diatas adalah :



```
C:\Windows\system32\cmd.exe
Pegawai
Nama
ID
```

Jika anda perhatikan pada hasil diatas maka dilayar akan dituliskan nama dari elemen yang terkait (objek PegawaiPertama)

XNode.NodesAfterSelf()

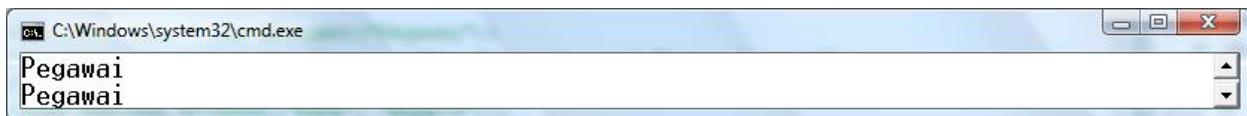
Prosedur ini akan mengembalikan semua Node yang ada setelah objek terkait dalam dokumen XML. Perlu diingat bahwa Node yang akan dikembalikan hanyalah Node yang berada satu level dengan Node yang dikenakan prosedur ini. Contoh kode penggunaan prosedur ini adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama = new XElement("Pegawai",
        new XComment("Ini Pegawai Pertama"),
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11"))

    ,
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))

    ,
    new XElement("Pegawai",
        new XElement("Nama", "Narendra"),
        new XElement("ID", "2"))
));
foreach ( XElement hasil in PegawaiPertama.NodesAfterSelf())
    Console.WriteLine(hasil.Name);
```

Hasil dari kode diatas adalah :



```
C:\Windows\system32\cmd.exe
Pegawai
Pegawai
```

Terlihat dalam hasil bahwa dituliskan dua buah nama dari Node yang berada setelah Node PegawaiPertama dan juga dalam level yang sama pada sebuah dokumen XML.

XNode.ElementsAfterSelf()

Berbeda dengan prosedur NodeAfterSelf() yang akan mengembalikan semua Node yang ada setelah Node yang terkait. Fungsi ini secara khusus hanya akan mengembalikan semua Element yang ada setelah Node yang dikenakan operasi ini . Contoh kode penggunaan prosedur ini adalah :

```
XElement NamaPegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    new XElement("Pegawai",
        NamaPegawaiPertama = new XElement("Nama", "Ronald"),
        new XComment("Ini Pegawai Pertama"),
        new XElement("ID", "11"))
```

```

        ,
        new XElement("Pegawai",
            new XElement("Nama", "Fajar"),
            new XElement("ID", "1"))
    );
    foreach (var hasil in NamaPegawaiPertama.ElementsAfterSelf())
        Console.WriteLine(hasil);

```

Hasil dari kode diatas adalah :



```
<ID>11</ID>
```

Terlihat bahwa dilayar hanya akan ditampilkan nilai dari Node yang bertipe XElement. Jika kita mengganti prosedur ElementsAfterSelf menjadi NodesAfterSelf seperti dibawah ini maka hasil dari kode kita akan berubah :

```

 XElement NamaPegawaiPertama;
 XDocument doc = new XDocument(new XElement("KumpulanPegawai",
     new XElement("Pegawai",
         NamaPegawaiPertama = new XElement("Nama", "Ronald"),
         new XComment("Ini Pegawai Pertama"),
         new XElement("ID", "11"))

     ,
     new XElement("Pegawai",
         new XElement("Nama", "Fajar"),
         new XElement("ID", "1"))
    );
    foreach (var hasil in NamaPegawaiPertama.NodesAfterSelf())
        Console.WriteLine(hasil);

```

Hasil dari kode diatas :



```
<!--Ini Pegawai Pertama-->
<ID>11</ID>
```

Terlihat bahwa sekarang node yang bertipe XComment juga ditampilkan dilayar.

XNode.NodesBeforeSelf()

Prosedur ini akan mengembalikan semua Node yang ada sebelum objek terkait dalam dokumen XML. Perlu diingat bahwa Node yang akan dikembalikan hanyalah Node yang berada satu level dengan Node yang dikenakan prosedur ini. Contoh kode penggunaan prosedur ini adalah :

```

 XElement PegawaiKetiga;
 XDocument doc = new XDocument(new XElement("KumpulanPegawai",
     new XElement("Pegawai",
         new XElement("Nama", "Ronald"),
         new XComment("Ini Pegawai Pertama"),
         new XElement("ID", "11"))

     ,
     new XElement("Pegawai",

```

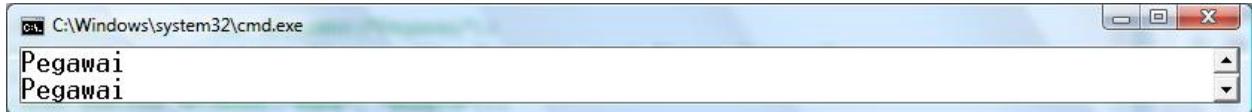
```

        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))

        PegawaiKetiga = new XElement("Pegawai",
            new XElement("Nama", "Narendra"),
            new XElement("ID", "2"))
    );
foreach ( XElement hasil in PegawaiKetiga.NodesBeforeSelf())
    Console.WriteLine(hasil.Name);

```

Hasil dari kode diatas adalah :



```
C:\Windows\system32\cmd.exe
Pegawai
Pegawai
```

Terlihat dalam hasil bahwa dituliskan dua buah nama dari Node yang berada sebelum Node PegawaiKetiga dan juga dalam level yang sama pada sebuah dokumen XML.

XNode.ElementsBeforeSelf()

Berbeda dengan prosedur NodeBeforeSelf() yang akan mengembalikan semua Node yang ada sebelum Node yang terkait. Fungsi ini secara khusus hanya akan mengembalikan semua Element yang ada sebelum Node yang dikenakan operasi ini . Contoh kode penggunaan prosedur ini adalah :

```

 XElement IDPegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    new XElement("Pegawai",
        new XElement("Nama", "Ronald"),
        new XComment("Ini Pegawai Pertama"),
        IDPegawaiPertama = new XElement("ID", "11"))

    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))

    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))
));
foreach ( XElement hasil in IDPegawaiPertama.ElementsBeforeSelf())
    Console.WriteLine(hasil);

```

Hasil dari kode diatas adalah :



```
C:\Windows\system32\cmd.exe
<Nama>Ronald</Nama>
```

Terlihat bahwa dilayar hanya akan ditampilkan nilai dari Node yang bertipe XElement. Jika kita mengganti prosedur ElementsBeforeSelf menjadi NodesBeforeSelf seperti dibawah ini maka hasil dari kode kita akan berubah :

```

 XElement IDPegawaiPertama;
 XDocument doc = new XDocument(new XElement("KumpulanPegawai",
     new XElement("Pegawai",
         new XElement("Nama", "Ronald"),
         new XComment("Ini Pegawai Pertama"),
         IDPegawaiPertama = new XElement("ID", "11"))

     new XElement("Pegawai",
         new XElement("Nama", "Fajar"),
         new XElement("ID", "1"))

     new XElement("Pegawai",
         new XElement("Nama", "Fajar"),
         new XElement("ID", "1"))
    ));
 foreach (var hasil in IDPegawaiPertama.NodesBeforeSelf())
    Console.WriteLine(hasil);Hasil dari kode diatas :

```

The screenshot shows a command-line interface window titled 'cmd C:\Windows\system32\cmd.exe'. The window displays the following XML content:

```

<Nama>Ronald</Nama>
<!--Ini Pegawai Pertama-->

```

Terlihat bahwa sekarang node yang bertipe XComment juga ditampilkan dilayar.

Modifikasi XML

Dengan menggunakan API LINQ to SQL operasi modifikasi terhadap sebuah dokumen XML akan terasa lebih mudah. Hal ini akan saya coba tunjukkan dengan penjelasan beberapa prosedur yang akan berguna ketika anda melakukan penambahan,perubahan, ataupun penghapusan sebuah elemen atau node.

Menambahkan sebuah Node baru

Anda dapat melakukan penambahan sebuah Node dengan menggunakan 4 prosedur yang ada dalam kelas XContainer.

XContainer.Add()

Prosedur ini akan menambahkan Node pada pohon XML sebagai anak dari objek XContainer yang dikenakan operasi ini. Node anak akan ditambahkan sebagai anak terakhir dari objek XContainer. Contoh kode penggunaanya adalah sebagai berikut :

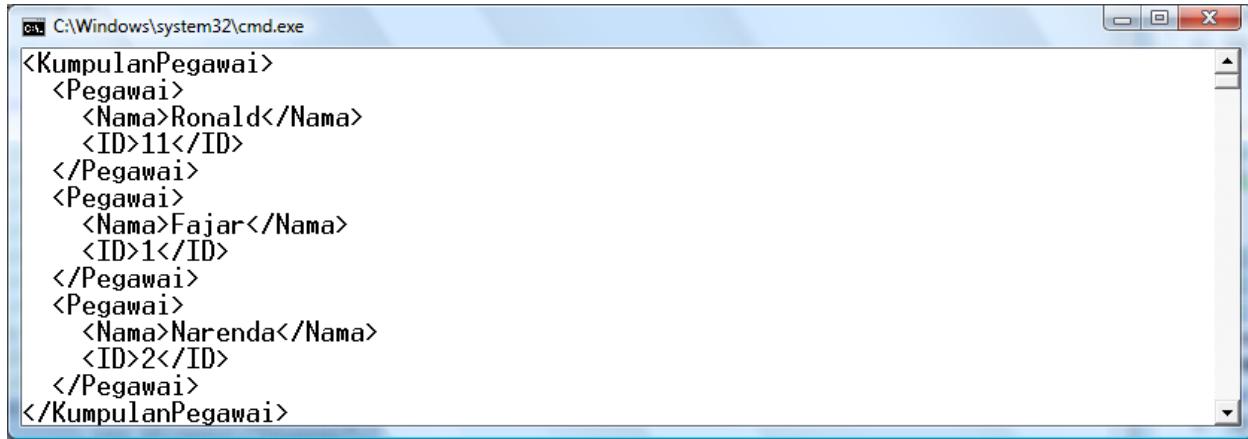
```

XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    new XElement("Pegawai",
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11")),
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))
));
doc.Element("KumpulanPegawai").Add(
    new XElement("Pegawai",
        new XElement("Nama", "Narendra"),
        new XElement("ID", "2"))
);

```

```
Console.WriteLine(doc);
```

Terlihat dari kode diatas bahwa ditambahkan sebuah XElement kepada sebuah objek XElement dengan nama "KumpulanPegawai" maka objek XElement yang ditambahkan akan ada dalam urutan terakhir dari kumpulan anak. Hasil dari kode diatas dapat dilihat dibawah ini :



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The output displayed is an XML document structure:

```
<KumpulanPegawai>
  <Pegawai>
    <Nama>Ronald</Nama>
    <ID>11</ID>
  </Pegawai>
  <Pegawai>
    <Nama>Fajar</Nama>
    <ID>1</ID>
  </Pegawai>
  <Pegawai>
    <Nama>Narendra</Nama>
    <ID>2</ID>
  </Pegawai>
</KumpulanPegawai>
```

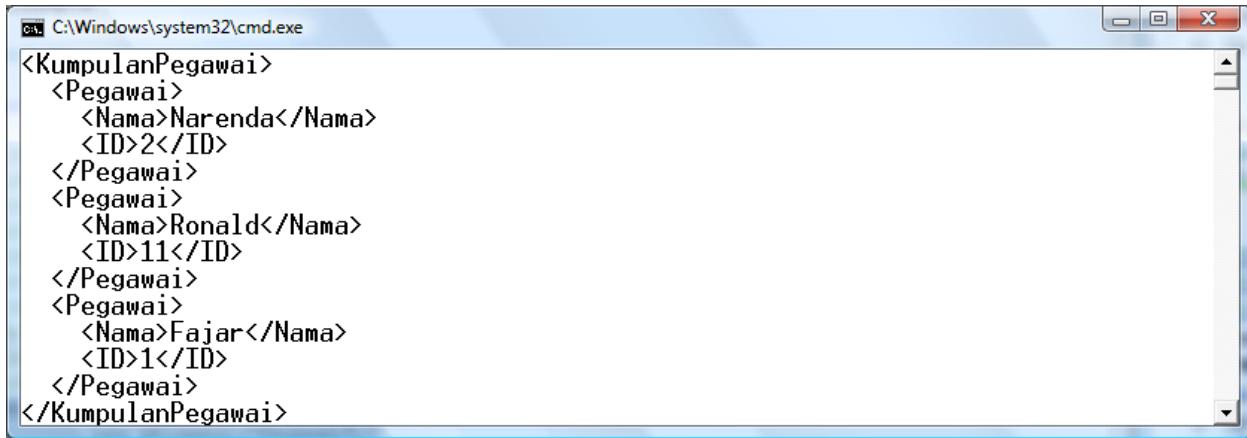
Terlihat bahwa XElement yang ditambahkan ada di urutan terakhir

XContainer.AddFirst()

Fungsionalitas dari prosedur ini sama dengan prosedur Add() yaitu akan menambahkan Node sebagai anak. Namun jika pada prosedur Add() Node akan ditambahkan sebagai anak yang terakhir maka jika menggunakan prosedur ini maka Node yang baru akan ditambahkan sebagai anak yang pertama. Contoh kode penggunaan prosedur ini adalah :

```
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    new XElement("Pegawai",
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11")),
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))
));
doc.Element("KumpulanPegawai").AddFirst(
    new XElement("Pegawai",
        new XElement("Nama", "Narendra"),
        new XElement("ID", "2"))
);
Console.WriteLine(doc);
```

Hasil dari kode diatas adalah :



```
C:\Windows\system32\cmd.exe
<KumpulanPegawai>
  <Pegawai>
    <Nama>Narendra</Nama>
    <ID>2</ID>
  </Pegawai>
  <Pegawai>
    <Nama>Ronald</Nama>
    <ID>11</ID>
  </Pegawai>
  <Pegawai>
    <Nama>Fajar</Nama>
    <ID>1</ID>
  </Pegawai>
</KumpulanPegawai>
```

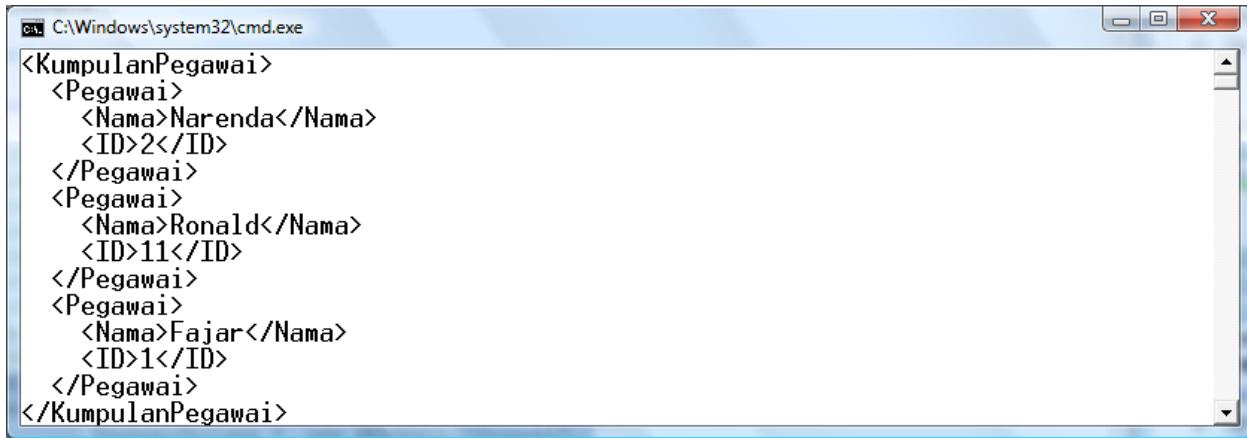
Dari hasil diatas terlihat jelas bahwa XElement Pegawai yang baru akan ditambahkan sebagai anak yang pertama dari XElement "KumpulanPegawai"

XContainer.AddBeforeSelf()

Prosedur ini akan menambahkan Node sebelum XContainer yang dikenakan operasi. Namun berbeda dengan prosedur Add() dan AddFirst(), prosedur ini akan menambahkan Node sebagai "sibling" bukan sebagai anak. Sehingga Node yang baru akan berada selevel dengan XContainer yang dikenakan prosedur ini. Contoh penggunaan prosedur ini adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama = new XElement("Pegawai",
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11")),
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))
));
PegawaiPertama.AddBeforeSelf(
    new XElement("Pegawai",
        new XElement("Nama", "Narendra"),
        new XElement("ID", "2"))
);
Console.WriteLine(doc);
```

Hasil dari kode diatas adalah :



```
C:\Windows\system32\cmd.exe
<KumpulanPegawai>
  <Pegawai>
    <Nama>Narendra</Nama>
    <ID>2</ID>
  </Pegawai>
  <Pegawai>
    <Nama>Ronald</Nama>
    <ID>11</ID>
  </Pegawai>
  <Pegawai>
    <Nama>Fajar</Nama>
    <ID>1</ID>
  </Pegawai>
</KumpulanPegawai>
```

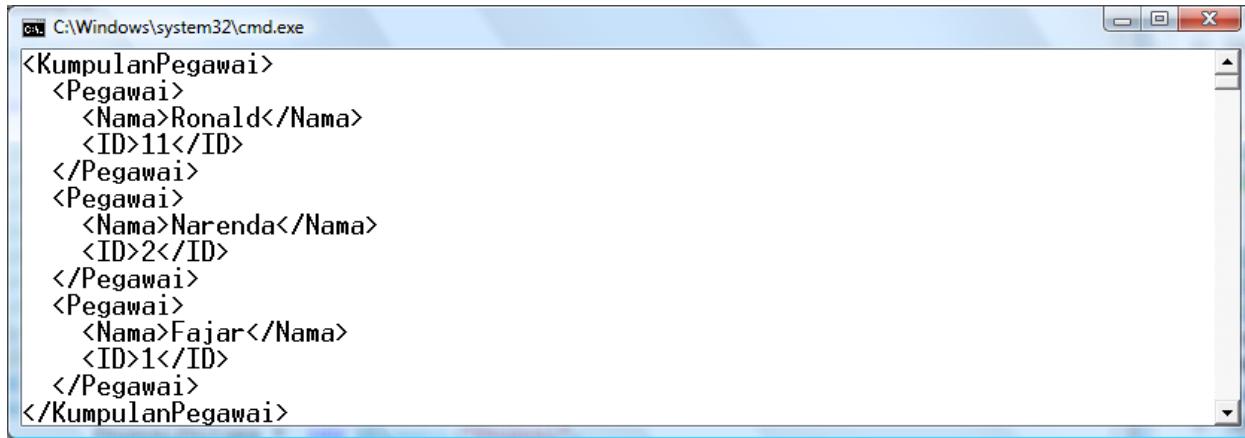
Terlihat bahwa XElement Pegawai yang mengandung Element Nama dengan nilai "Narendra" akan ditambahkan sebelum objek PegawaiPertama (objek yang mengandung Element Nama dengan nilai "Ronald"). XElement yang baru ditambahkan tersebut pun ditambahkan bukan sebagai elemen anak melainkan sebagai elemen yang memiliki tingkatan yang sama dalam pohon XML atau sebagai sibling.

XContainer.AddAfterSelf()

Jika pada prosedur AddBeforeSelf() Node yang baru akan ditambahkan sebelum XContainer yang dikenakan prosedur, maka pada prosedur ini Node yang baru akan ditambahkan setelah objek XContainer yang dikenakan operasi. Contoh kode penggunaan prosedur ini adalah

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama = new XElement("Pegawai",
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11")),
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))
));
PegawaiPertama.AddAfterSelf(
    new XElement("Pegawai",
        new XElement("Nama", "Narendra"),
        new XElement("ID", "2")))
);
Console.WriteLine(doc);
```

Hasil dari kode diatas adalah :



```
C:\Windows\system32\cmd.exe
<KumpulanPegawai>
  <Pegawai>
    <Nama>Ronald</Nama>
    <ID>11</ID>
  </Pegawai>
  <Pegawai>
    <Nama>Narendra</Nama>
    <ID>2</ID>
  </Pegawai>
  <Pegawai>
    <Nama>Fajar</Nama>
    <ID>1</ID>
  </Pegawai>
</KumpulanPegawai>
```

Terlihat bahwa XElement Pegawai yang mengandung Element Nama dengan nilai "Narendra" akan ditambahkan setelah objek PegawaiPertama (objek yang mengandung Element Nama dengan nilai "Ronald"). XElement yang baru ditambahkan tersebut pun ditambahkan bukan sebagai elemen anak melainkan sebagai elemen yang memiliki tingkatan yang sama dalam pohon XML atau sebagai sibling.

Menghapus sebuah Node

Anda dapat melakukan penghapusan sebuah Node dengan menggunakan baik prosedur Remove maupun RemoveAll seperti yang akan dijelaskan pada bagian berikut :

XNode.Remove()

Prosedur ini akan menghapus Node yang dikenakan prosedur dari sebuah pohon XML. Contoh penggunaan kode ini adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama = new XElement("Pegawai",
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11")),
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))
));
Console.WriteLine(doc);
Console.ForegroundColor = ConsoleColor.Red;
Console.WriteLine("====Melakukan Operasi Penghapusan====");
Console.ForegroundColor = ConsoleColor.Black;
PegawaiPertama.Remove();
Console.WriteLine(doc);
```

Hasil dari kode diatas adalah :

```
C:\Windows\system32\cmd.exe
<KumpulanPegawai>
  <Pegawai>
    <Nama>Ronald</Nama>
    <ID>11</ID>
  </Pegawai>
  <Pegawai>
    <Nama>Fajar</Nama>
    <ID>1</ID>
  </Pegawai>
</KumpulanPegawai>
=====Melakukan Operasi Penghapusan=====
<KumpulanPegawai>
  <Pegawai>
    <Nama>Fajar</Nama>
    <ID>1</ID>
  </Pegawai>
</KumpulanPegawai>
```

Terlihat dalam hasil diatas bahwa sebelum tulisan “=====Melakukan Operasi Penghapusan====” Elemen pegawai yang mengandung elemen Nama dengan nilai “Ronald” masih ada. Namun pada bagian setelah tulisan “=====Melakukan Operasi Penghapusan====” Elemen tersebut telah hilang dari pohon XML.

IEnumerable<T>.Remove()

Prosedur ini akan menghapus sebuah Node dari sebuah koleksi IEnumerable<T>. Contoh penggunaan prosedur ini adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama = new XElement("Pegawai",
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11")),
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))
));
Console.WriteLine(doc);
Console.ForegroundColor = ConsoleColor.Red;
Console.WriteLine("=====Melakukan Operasi Penghapusan====");
Console.ForegroundColor = ConsoleColor.Black;
doc.Descendants("ID").Remove();
Console.WriteLine(doc);
```

Kode diatas akan menghapus semua XElement yang memiliki Nama “ID” dari pohon XML, sehingga hasil dari kode diatas adalah :

```
C:\Windows\system32\cmd.exe
<KumpulanPegawai>
  <Pegawai>
    <Nama>Ronald</Nama>
    <ID>11</ID>
  </Pegawai>
  <Pegawai>
    <Nama>Fajar</Nama>
    <ID>1</ID>
  </Pegawai>
</KumpulanPegawai>
=====Melakukan Operasi Penghapusan=====
<KumpulanPegawai>
  <Pegawai>
    <Nama>Ronald</Nama>
  </Pegawai>
  <Pegawai>
    <Nama>Fajar</Nama>
  </Pegawai>
</KumpulanPegawai>
```

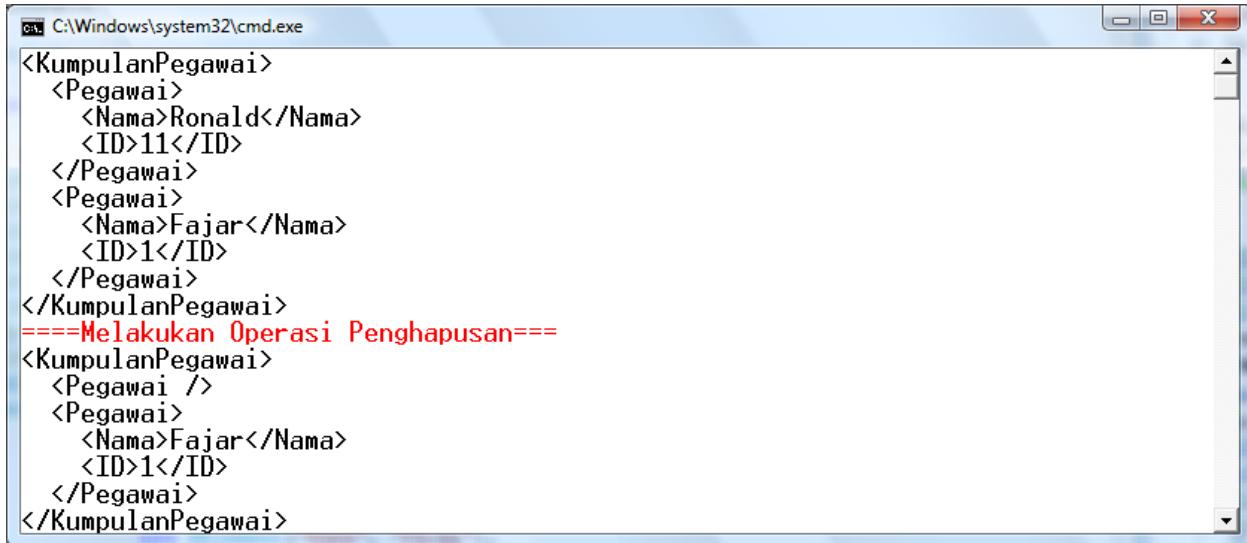
Terlihat dalam hasil diatas bahwa sebelum tulisan “=====Melakukan Operasi Penghapusan====” semua elemen dengan nama “ID” masih ada. Namun pada bagian setelah tulisan “=====Melakukan Operasi Penghapusan====” semua elemen tersebut telah hilang dari pohon XML.

XElement.RemoveAll()

Prosedur ini akan menghapus hanya isi dari sebuah XElement. Prosedur ini cocok dipakai jika misalnya anda hanya ingin menghapus isi dari sebuah elemen tapi tidak ingin menghapus elemen tersebut. Contoh kode penggunaan prosedur ini adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama = new XElement("Pegawai",
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11")),
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))
));
Console.WriteLine(doc);
Console.ForegroundColor = ConsoleColor.Red;
Console.WriteLine("=====Melakukan Operasi Penghapusan====");
Console.ForegroundColor = ConsoleColor.Black;
PegawaiPertama.RemoveAll();
Console.WriteLine(doc);
```

Kode diatas akan menghapus semua objek yang ada dalam XElement PegawaiPertama, baik itu berupa atribut maupun elemen anak. Hasil dari kode diatas adalah :



```
C:\Windows\system32\cmd.exe
<KumpulanPegawai>
  <Pegawai>
    <Nama>Ronald</Nama>
    <ID>11</ID>
  </Pegawai>
  <Pegawai>
    <Nama>Fajar</Nama>
    <ID>1</ID>
  </Pegawai>
</KumpulanPegawai>
=====Melakukan Operasi Penghapusan=====
<KumpulanPegawai>
  <Pegawai />
  <Pegawai>
    <Nama>Fajar</Nama>
    <ID>1</ID>
  </Pegawai>
</KumpulanPegawai>
```

Terlihat bahwa sekarang XElemen PegawaiPertama tidak berisikan apapun.

Mengupdate sebuah Node

Beberapa kelas turunan dari XNode seperti XElement,XComment memiliki sebuah property Value yang dapat secara langsung diupdate. Sedangkan kelas lainnya seperti XDeclaration memiliki properti spesifik yang juga bisa diupdate. Khusus untuk element kita dapat menggunakan XElement.SetValue atau XContainer.ReplaceAll yang akan dibahas juga pada bagian ini.

Penggunaan properti Value pada XElement dan XComment.

Seperti telah disebutkan sebelumnya bahwa kita dapat menggunakan properti Value pada setiap objek dari kelas XNode. Sehingga kita dapat mengganti-ganti nilai dari objek ini dengan mengeset nilai Valuenya. Contoh penggunaannya adalah pada kode dibawah ini :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama = new XElement("Pegawai",
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11")),
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XElement("ID", "1"))
));
Console.WriteLine(doc);
Console.ForegroundColor = ConsoleColor.Red;
Console.WriteLine("=====Melakukan Operasi Penggantian====");
Console.ForegroundColor = ConsoleColor.Black;
PegawaiPertama.Element("Nama").Value = "Joko";
Console.WriteLine(doc);
```

Kode diatas akan mengganti nilai element Nama dari “Ronald” menjadi “Joko”. Hasil dari kode diatas adalah :

```
C:\Windows\system32\cmd.exe
<KumpulanPegawai>
  <Pegawai>
    <Nama>Ronald</Nama>
    <ID>11</ID>
  </Pegawai>
  <Pegawai>
    <Nama>Fajar</Nama>
    <ID>1</ID>
  </Pegawai>
</KumpulanPegawai>
=====Melakukan Operasi Penggantian=====
<KumpulanPegawai>
  <Pegawai>
    <Nama>Joko</Nama>
    <ID>11</ID>
  </Pegawai>
  <Pegawai>
    <Nama>Fajar</Nama>
    <ID>1</ID>
  </Pegawai>
</KumpulanPegawai>
```

Penggunaan XContainer.ReplaceAll()

Prosedur ini berguna jika anda ingin mengganti seluruh elemen anak dari sebuah Node. Contoh penggunaan kode ini adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama= new XElement("Pegawai",
        new XElement("Nama", "Ronald"),
        new XAttribute("ID", "11")),
    new XElement("Pegawai",
        new XElement("Nama", "Fajar"),
        new XAttribute("ID", "1"))
));
Console.WriteLine(doc);
Console.ForegroundColor = ConsoleColor.Red;
Console.WriteLine("=====Melakukan Operasi Penggantian====");
Console.ForegroundColor = ConsoleColor.Black;
PegawaiPertama.ReplaceAll(new XElement("Umur", "21"));
Console.WriteLine(doc);
```

Kode diatas akan mengganti seluruh elemen anak dari setiap objek XElement yang memiliki nama Pegawai. Hasil dari kode diatas adalah dibawah ini:

```
C:\Windows\system32\cmd.exe
<KumpulanPegawai>
  <Pegawai ID="11">
    <Nama>Ronald</Nama>
  </Pegawai>
  <Pegawai ID="1">
    <Nama>Fajar</Nama>
  </Pegawai>
</KumpulanPegawai>
=====Melakukan Operasi Penggantian=====
<KumpulanPegawai>
  <Pegawai>
    <Umur>21</Umur>
  </Pegawai>
  <Pegawai ID="1">
    <Nama>Fajar</Nama>
  </Pegawai>
</KumpulanPegawai>
```

Jika anda jeli maka terlihat juga bahwa atribut di XElement yang memiliki nama “Pegawai” pun ikut hilang. Padahal atribut (diwakili oleh kelas XAttribute) bukan merupakan anak dari XElement, namun turut juga dihapus.

Penggunaan XElement.SetElementValue()

Prosedur ini menurut saya sangat-sangat menarik. Memang namanya sangat sederhana namun dibalik kesederhanaannya ini sebenarnya prosedur ini cukup powerfull untuk melakukan penambahan,perubahan maupun penghapusan sebuah elemen anak dari objek XElement. Prosedur ini akan meminta 2 buah parameter masukan, parameter yang pertama berupa nama dari elemen anak yang ingin anda masukkan nilainya dan yang kedua adalah nilai dari elemen anak tersebut. Perilaku dari prosedur ini adalah :

1. Jika nama elemen anak yang ingin anda masukkan tersebut telah ada dalam daftar anak objek XElement maka nilai elemen anak tersebut akan diganti dengan nilai yang baru selama nilai elemen anak yang baru tersebut tidak null. Sehingga kita mendapatkan fungsi update dengan menggunakan perilaku ini, contoh kode penggunaannya adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama= new XElement("Pegawai",
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11"))));
Console.WriteLine(doc);
Console.ForegroundColor = ConsoleColor.Red;
Console.WriteLine("=====Melakukan Operasi Penggantian====");
Console.ForegroundColor = ConsoleColor.Black;
PegawaiPertama.SetElementValue("Nama", "Joko");
Console.WriteLine(doc);
```

Hasil dari kode diatas dapat dilihat pada gambar dibawah. Terlihat jelas bahwa nilai dari elemen yang bernama “Nama” telah diganti

```
C:\Windows\system32\cmd.exe
<KumpulanPegawai>
  <Pegawai>
    <Nama>Ronald</Nama>
    <ID>11</ID>
  </Pegawai>
</KumpulanPegawai>
====Melakukan Operasi Penggantian===
<KumpulanPegawai>
  <Pegawai>
    <Nama>Joko</Nama>
    <ID>11</ID>
  </Pegawai>
</KumpulanPegawai>
```

2. Jika nama dari elemen anak yang ingin anda masukkan tersebut tidak ada dalam daftar anak objek XElement maka akan ditambahkan sebuah elemen anak baru dengan nama dan nilai sesuai dengan yang ada di parameter masukan prosedur ini. Sehingga kita mendapatkan fungsi add dengan menggunakan perilaku ini, contoh kode penggunaannya adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama= new XElement("Pegawai",
        new XElement("Nama", "Ronald"),
        new XElement("ID", "11"))));
Console.WriteLine(doc);
Console.ForegroundColor = ConsoleColor.Red;
Console.WriteLine("====Melakukan Operasi Penggantian===");
Console.ForegroundColor = ConsoleColor.Black;
PegawaiPertama.SetElementValue("Umur", "21");
Console.WriteLine(doc);
```

Hasil dari kode diatas dapat dilihat pada gambar dibawah. Terlihat jelas bahwa karena belum ada elemen dengan nama "Umur" maka sebuah elemen baru akan ditambahkan.

```
C:\Windows\system32\cmd.exe
<KumpulanPegawai>
  <Pegawai>
    <Nama>Ronald</Nama>
    <ID>11</ID>
  </Pegawai>
</KumpulanPegawai>
====Melakukan Operasi Penggantian===
<KumpulanPegawai>
  <Pegawai>
    <Nama>Ronald</Nama>
    <ID>11</ID>
    <Umur>21</Umur>
  </Pegawai>
</KumpulanPegawai>
```

3. Jika nama dari elemen anak yang ingin anda masukkan tersebut telah ada dalam daftar anak objek XElement dan nilai elemen anak tersebut null maka elemen anak dengan nama yang sama dan telah ada sebelumnya akan dihapus. Sehingga kita mendapatkan fungsi delete dengan menggunakan perilaku ini, contoh kode penggunaannya adalah :

```

 XElement PegawaiPertama;
 XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama= new XElement("Pegawai",
       new XElement("Nama", "Ronald"),
       new XElement("ID", "11"))));
 Console.WriteLine(doc);
 Console.ForegroundColor = ConsoleColor.Red;
 Console.WriteLine("====Melakukan Operasi Penggantian===");
 Console.ForegroundColor = ConsoleColor.Black;
 PegawaiPertama.SetElementValue("ID", null);
 Console.WriteLine(doc);

```

Hasil dari kode diatas dapat dilihat pada gambar dibawah ini. Terlihat bahwa karena elemen dengan nama "ID" telah ada sebelumnya maka elemen tersebut akan dihapus, karena nilai masukannya adalah null.

```

<KumpulanPegawai>
  <Pegawai>
    <Nama>Ronald</Nama>
    <ID>11</ID>
  </Pegawai>
</KumpulanPegawai>
=====Melakukan Operasi Penggantian=====
<KumpulanPegawai>
  <Pegawai>
    <Nama>Ronald</Nama>
  </Pegawai>
</KumpulanPegawai>

```

Operasi pada Atribut XML

Pada bagian ini kita akan melihat bagaimana mudahnya melakukan operasi penelusuran dan juga modifikasi sebuah atribut dalam dokumen XML.

Penelusuran Atribut

Penelusuran atribut dapat dilakukan dengan menggunakan properti dan juga fungsi yang akan dijelaskan pada bagian ini.

XElement.FirstAttribute

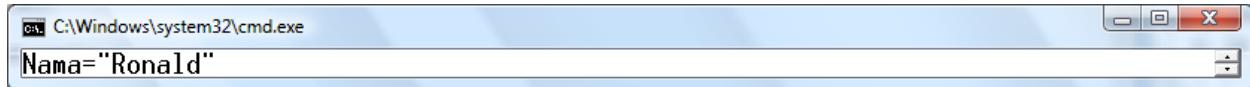
FirstAttribute merupakan properti pada XElement yang akan berisikan sebuah atribut pertama yang ada dalam Elemen tersebut. Contoh kode penggunaannya adalah :

```

 XElement PegawaiPertama;
 XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama= new XElement("Pegawai",
       new XAttribute("Nama", "Ronald"),
       new XAttribute("ID", "11"))));
 Console.WriteLine(PegawaiPertama.FirstAttribute);

```

Hasil dari kode diatas akan menuliskan pasangan nama dan nilai dari atribut pertama yang ditemui :



```
C:\Windows\system32\cmd.exe
Nama="Ronald"
```

XAttribute.NextAttribute

NextAttribute merupakan sebuah properti pada XAttribute yang akan menunjuk ke atribut yang ada setelah atribut ini dalam elemen yang sama. Contoh kode penggunaanya adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama= new XElement("Pegawai",
        new XAttribute("Nama", "Ronald"),
        new XAttribute("ID", "11"))));
Console.WriteLine(PegawaiPertama.FirstAttribute.NextAttribute);
```

Hasil dari kode diatas akan menuliskan pasangan nama dan nilai dari atribut ID yang ada setelah atribut dengan nama “Nama”



```
C:\Windows\system32\cmd.exe
ID="11"
```

XElement.Attribute()

Prosedur ini akan mengembalikan sebuah atribut yang namanya sama dengan parameter masukan. Contoh penggunaan prosedur ini adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama= new XElement("Pegawai",
        new XAttribute("Nama", "Ronald"),
        new XAttribute("ID", "11"))));
Console.WriteLine(PegawaiPertama.Attribute("Nama"));
```

Hasil dari kode diatas akan menampilkan sebuah pasangan nama dan nilai dari attribut dengan nama “Nama”



```
C:\Windows\system32\cmd.exe
Nama="Ronald"
```

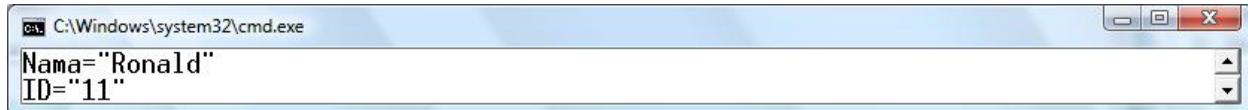
XElement.Attributes()

Prosedur ini akan mengembalikan sebuah koleksi yang berisikan atribut –atribut yang ada pada elemen terkait. Contoh kode penggunaan prosedur ini adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama= new XElement("Pegawai",
        new XAttribute("Nama", "Ronald"),
        new XAttribute("ID", "11"))));

foreach (var att in PegawaiPertama.Attributes())
    Console.WriteLine(att);
```

Kode diatas akan menuliskan setiap pasangan nama dan nilai dari attribut yang ada dalam objek XElement PegawaiPertama, sehingga hasil dari kode diatas adalah :



```
C:\Windows\system32\cmd.exe
Nama="Ronald"
ID="11"
```

Modifikasi Atribut

Modifikasi atribut dapat dilakukan dengan mengedit langsung nilai properti Value dari XAttribute maupun menggunakan Prosedur SetAttributeValue yang ada pada elemen XElement seperti yang dicontohkan pada bagian berikut

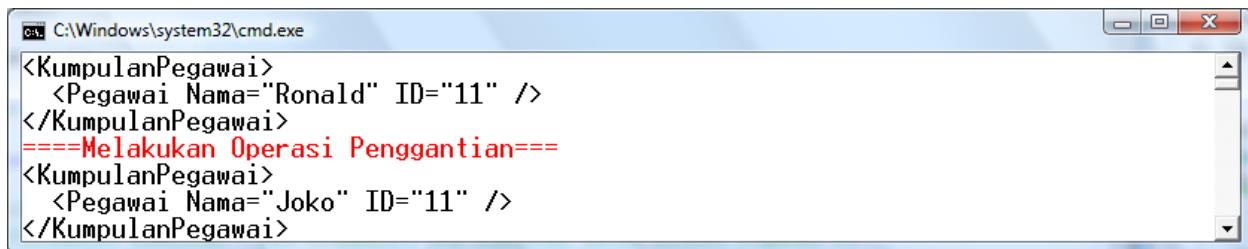
XAttribute.Value

Properti ini merupakan properti yang menunjuk pada nilai value dari sebuah XAttribute. Contoh penggunaan kode ini adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama= new XElement("Pegawai",
        new XAttribute("Nama", "Ronald"),
        new XAttribute("ID", "11"))));

Console.WriteLine(doc);
Console.ForegroundColor = ConsoleColor.Red;
Console.WriteLine("====Melakukan Operasi Penggantian===");
Console.ForegroundColor = ConsoleColor.Black;
PegawaiPertama.FirstAttribute.Value = "Joko";
```

Hasil dari kode diatas adalah:



```
C:\Windows\system32\cmd.exe
<KumpulanPegawai>
  <Pegawai Nama="Ronald" ID="11" />
</KumpulanPegawai>
====Melakukan Operasi Penggantian===
<KumpulanPegawai>
  <Pegawai Nama="Joko" ID="11" />
</KumpulanPegawai>
```

Terlihat bahwa nilai value dari atribut pada objek PegawaiPertama yang bernama "Nama" telah diubah dari "Ronald" menjadi "Joko"

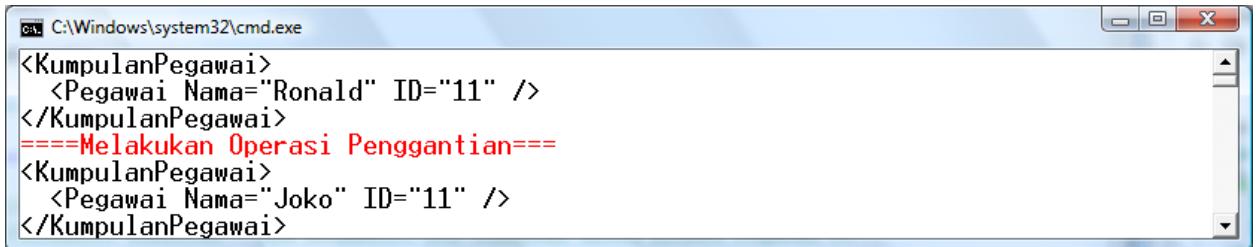
XElement.SetAttributeValue()

Tipe penggunaan dari prosedur ini mirip dengan XElement.SetValue. Memang namanya sangat sederhana namun dibalik kesederhanaannya ini sebenarnya prosedur ini cukup powerfull untuk melakukan penambahan,perubahan maupun penghapusan sebuah atribut dari objek XElement. Prosedur ini akan meminta 2 buah parameter masukan, parameter yang pertama berupa nama dari atribut yang ingin anda masukkan nilainya dan yang kedua adalah nilai dari atribut tersebut. Perilaku dari prosedur ini adalah :

1. Jika nama atribut yang ingin anda masukkan tersebut telah ada dalam daftar atribut objek XElement maka nilai atribut tersebut akan diganti dengan nilai yang baru selama nilai atribut yang baru tersebut tidak null. Sehingga kita mendapatkan fungsi update dengan menggunakan perilaku ini, contoh kode penggunaannya adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama= new XElement("Pegawai",
        new XAttribute("Nama", "Ronald"),
        new XAttribute("ID", "11"))));
Console.WriteLine(doc);
Console.ForegroundColor = ConsoleColor.Red;
Console.WriteLine("====Melakukan Operasi Penggantian===");
Console.ForegroundColor = ConsoleColor.Black;
PegawaiPertama.SetValue("Nama", "Joko");
Console.WriteLine(doc);
```

Hasil dari kode diatas dapat dilihat pada gambar dibawah. Terlihat jelas bahwa nilai dari atribut yang bernama "Nama" telah diganti



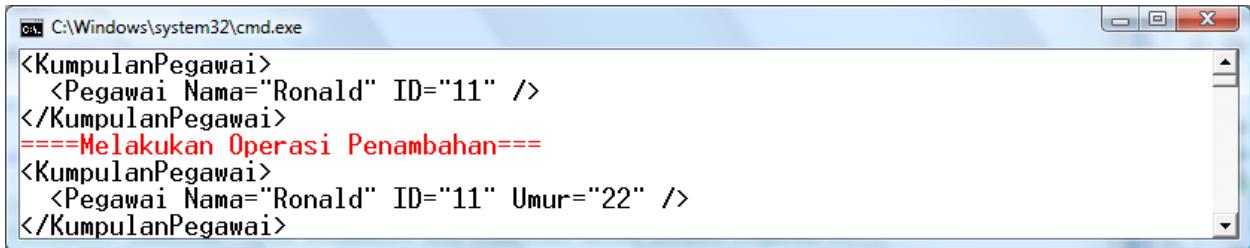
The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The window displays the following XML content:

```
<KumpulanPegawai>
  <Pegawai Nama="Ronald" ID="11" />
</KumpulanPegawai>
====Melakukan Operasi Penggantian===
<KumpulanPegawai>
  <Pegawai Nama="Joko" ID="11" />
</KumpulanPegawai>
```

2. Jika nama dari atribut yang ingin anda masukkan tersebut tidak ada dalam daftar atribut XElement maka akan ditambahkan sebuah atribut baru dengan nama dan nilai sesuai dengan yang ada di parameter masukan prosedur ini. Sehingga kita mendapatkan fungsi add dengan menggunakan perilaku ini, contoh kode penggunaannya adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama= new XElement("Pegawai",
        new XAttribute("Nama", "Ronald"),
        new XAttribute("ID", "11"))));
Console.WriteLine(doc);
Console.ForegroundColor = ConsoleColor.Red;
Console.WriteLine("====Melakukan Operasi Penambahan===");
Console.ForegroundColor = ConsoleColor.Black;
PegawaiPertama. SetAttributeValue ("Umur","22");
Console.WriteLine(doc);
```

Hasil dari kode diatas dapat dilihat pada gambar dibawah. Terlihat jelas bahwa karena belum ada atribut dengan nama "Umur" maka sebuah elemen baru akan ditambahkan.

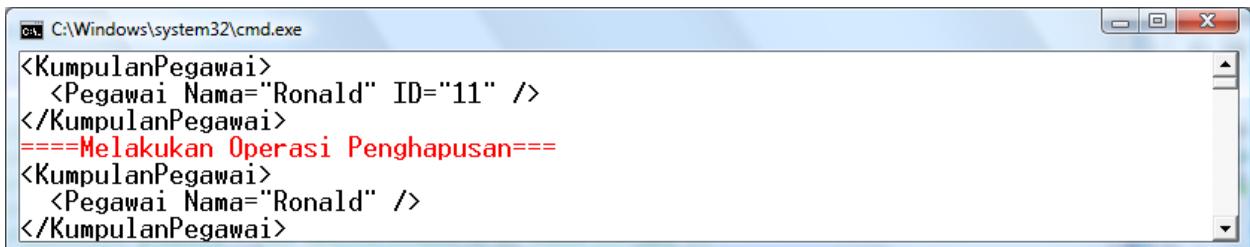


```
C:\Windows\system32\cmd.exe
<KumpulanPegawai>
  <Pegawai Nama="Ronald" ID="11" />
</KumpulanPegawai>
=====Melakukan Operasi Penambahan=====
<KumpulanPegawai>
  <Pegawai Nama="Ronald" ID="11" Umur="22" />
</KumpulanPegawai>
```

3. Jika nama dari atribut yang ingin anda masukkan tersebut telah ada dalam daftar attribut XElement dan nilai atribut tersebut null maka atribut dengan nama yang sama dan telah ada sebelumnya akan dihapus. Sehingga kita mendapatkan fungsi delete dengan menggunakan perilaku ini, contoh kode penggunaannya adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama= new XElement("Pegawai",
        new XAttribute("Nama", "Ronald"),
        new XAttribute ("ID", "11"))));
Console.WriteLine(doc);
Console.ForegroundColor = ConsoleColor.Red;
Console.WriteLine("=====Melakukan Operasi Penghapusan====");
Console.ForegroundColor = ConsoleColor.Black;
PegawaiPertama.SetAttributeValue("ID",null);
Console.WriteLine(doc);
```

Hasil dari kode diatas dapat dilihat pada gambar dibawah ini. Terlihat bahwa karena atribut dengan nama "ID" telah ada sebelumnya maka elemen tersebut akan dihapus, karena nilai masukannya adalah null.



```
C:\Windows\system32\cmd.exe
<KumpulanPegawai>
  <Pegawai Nama="Ronald" ID="11" />
</KumpulanPegawai>
=====Melakukan Operasi Penghapusan=====
<KumpulanPegawai>
  <Pegawai Nama="Ronald" />
</KumpulanPegawai>
```

Menghapus Atribut

Penghapusan dari atribut dapat dilakukan dengan menggunakan prosedur Remove() pada attribut yang bersangkutan. Namun jika anda ingin pula menghapus sebuah koleksi atribut anda dapat menggunakan prosedur Remove yang ada pada IEnumerable<T>.

XAttribute.Remove()

Prosedur ini akan menghapus atribut yang dikenakan prosedur dari sebuah pohon XML. Contoh penggunaan kode ini adalah :

```
XElement PegawaiPertama;
XDocument doc = new XDocument(new XElement("KumpulanPegawai",
```

```

PegawaiPertama= new XElement("Pegawai",
    new XAttribute("Nama", "Ronald"),
    new XAttribute("ID", "11")));

Console.WriteLine(doc);
Console.ForegroundColor = ConsoleColor.Red;
Console.WriteLine("====Melakukan Operasi Penghapusan===");
Console.ForegroundColor = ConsoleColor.Black;
PegawaiPertama.Attribute("Nama").Remove();
Console.WriteLine(doc);

```

Kode diatas akan menghapus sebuah atribut yang memiliki nama "Nama"

```

<KumpulanPegawai>
  <Pegawai Nama="Ronald" ID="11" />
</KumpulanPegawai>
====Melakukan Operasi Penghapusan===
<KumpulanPegawai>
  <Pegawai ID="11" />
</KumpulanPegawai>

```

IEnumerable<T>.Remove()

Prosedur ini digunakan ketika kita ingin menghapus sekumpulan atribut. Contoh kode penggunaan prosedur ini adalah :

```

 XElement PegawaiPertama;
 XDocument doc = new XDocument(new XElement("KumpulanPegawai",
    PegawaiPertama= new XElement("Pegawai",
        new XAttribute("Nama", "Ronald"),
        new XAttribute("ID", "11"))));

Console.WriteLine(doc);
Console.ForegroundColor = ConsoleColor.Red;
Console.WriteLine("====Melakukan Operasi Penghapusan===");
Console.ForegroundColor = ConsoleColor.Black;
PegawaiPertama.Attributes().Remove();
Console.WriteLine(doc);

```

Kode diatas akan menghapus semua atribut pada Elemen Pegawai. Hasil dari kode diatas dapat dilihat dibawah ini.

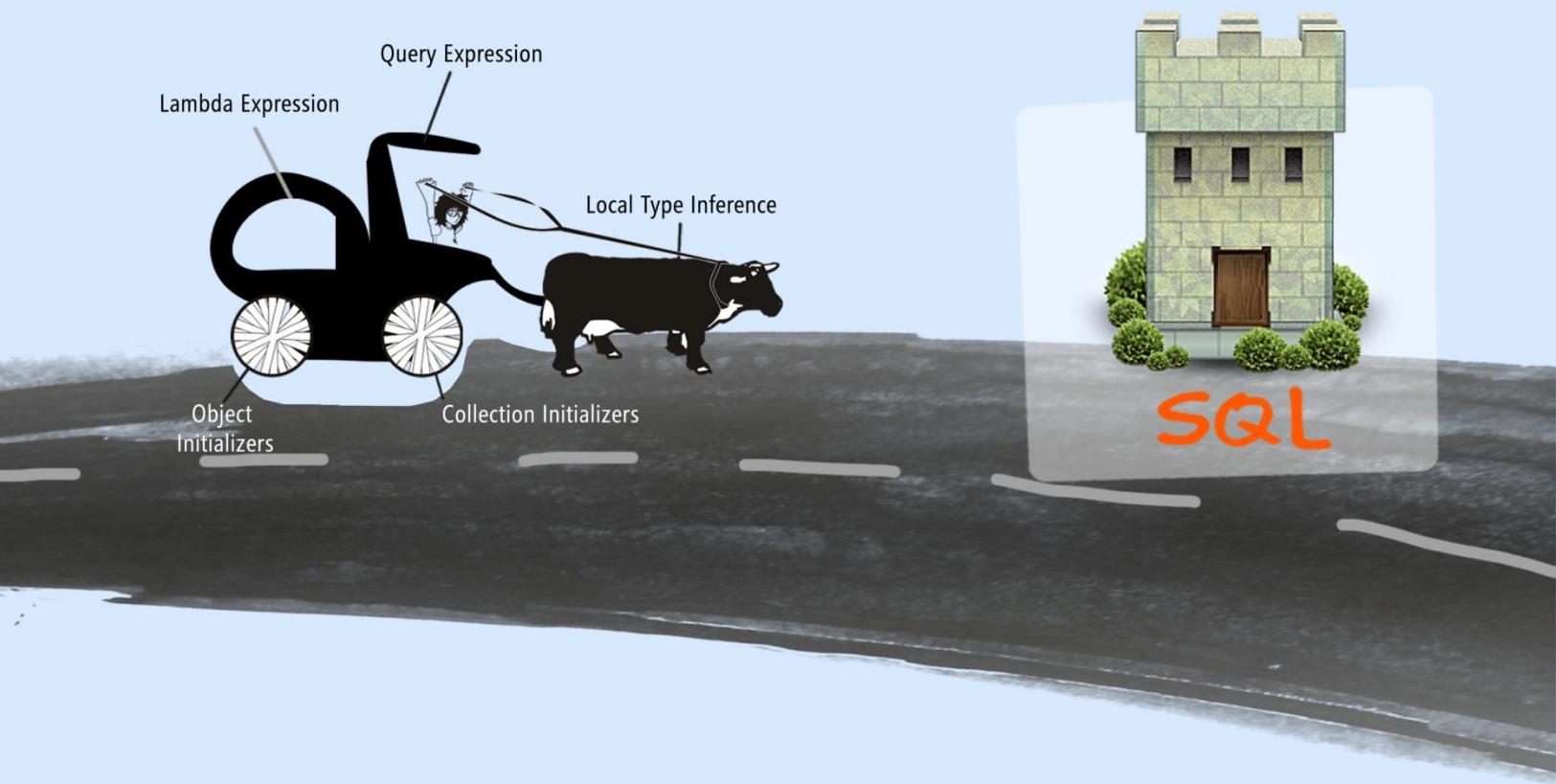
```

<KumpulanPegawai>
  <Pegawai Nama="Ronald" ID="11" />
</KumpulanPegawai>
====Melakukan Operasi Penghapusan===
<KumpulanPegawai>
  <Pegawai />
</KumpulanPegawai>

```

Penutup

Pada akhir perjalanan di desa LINQ to XML ini anda pasti telah melihat bagaimana kedigdayaan LINQ tetap berlanjut ketika sedang berurusan dengan format data XML. Cara pengaksesan yang sama dengan objek pun tetap berlaku dengan XML, sehingga seperti telah saya sebutkan diawal bahwa kita tidak perlu lagi mengembangkan skill khusus setiap kali berurusan dengan format data yang berbeda. Hanya saja tentunya ketika berhubungan dengan XML dengan menggunakan LINQ. Kita masih perlu memahami API dari LINQ to XML.



LINQ to SQL

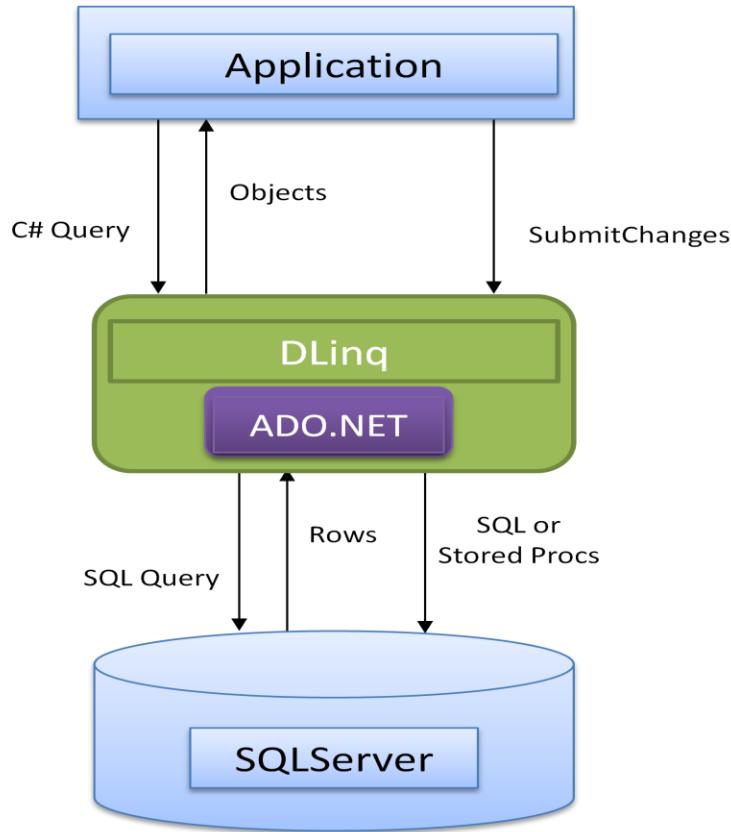
LINQ to SQL

Perjalanan kita sekarang sampai ke pondok terakhir yang akan dikunjungi dalam kesempatan kali ini yaitu pondok LINQ to SQL. Dari antara 3 tipe komponen LINQ yang dibahas di dalam buku ini, saya menilai bahwa LINQ to SQL adalah yang paling menarik. Mengapa paling menarik, karena jika anda membuka forum MSDN maupun milis netindonesia, komponen ini lah yang paling marak dibahas. Kemudian apa yang terbayang di pikiran anda para developer ketika mendengar kata data. Mayoritas dari developer kemungkinan besar akan berkata “Basisdata Relasional” dan LINQ to SQL hadir sebagai alternatif baru dalam hal ini.

Sebelum mulai banyak mengeksplorasi LINQ to SQL mari kita sejenak membahas mengenai bagaimana praktek pengembangan perangkat lunak yang menyangkut basis data selama ini dilakukan. Pada umumnya jika kita menggunakan bahasa yang berorientasi objek untuk membuat sebuah aplikasi, maka pada awalnya kita akan membuat sebuah kelas untuk memodelkan objek yang ada di dunia nyata. Ambil contoh pelanggan, pada umumnya kita akan memodelkan pelanggan sebagai sebuah kelas. Kemudian anggap pelanggan tersebut memiliki beberapa alamat dan juga beberapa nomor telepon yang bisa dihubungi. Tentunya dalam kelas pelanggan akan ada dua buah properti yang berupa koleksi (koleksi alamat dan nomor telepon).

Setelah itu tentunya aplikasi anda akan menyimpan objek dari kelas pelanggan tersebut agar dapat digunakan pada waktu yang lainnya. Penyimpanan bisa dilakukan dengan berbagai macam cara dan juga ke berbagai macam media penyimpanan, namun pada umumnya sekarang ini kita menggunakan basis data relasional dalam menyimpannya. Ketika kita menggunakan basis data relasional maka tentunya kita akan menyimpan objek tersebut ke lebih dari satu tabel (bayangkan 1 tabel untuk menyimpan properti pelanggan, 1 tabel untuk menyimpan alamat dan 1 lagi untuk menyimpan nomor telepon) kemudian kita pun perlu menerjemahkan tipe data yang kita gunakan pada bahasa pemrogramman agar sesuai dengan yang digunakan dalam basis data.

Langkah-langkah yang cukup merepotkan bukan? Permasalahan ini sering dikenal dengan istilah Object-relasional Impedance Mismatch, namun sekarang ini sudah banyak tools yang ada untuk menyelesaikan masalah ini dan LINQ to SQL pun hadir untuk menyelesaikan masalah ini, namun dengan tambahan integrasi yang kuat dengan bahasa pemrogramman. Secara sederhana LINQ to SQL digambarkan pada gambar dibawah ini .



Dalam gambar ini terlihat bahwa *query* dari bahasa C# akan diproses oleh lapisan Dlinq (LINQ to SQL) untuk diterjemahkan menjadi *Query SQL* untuk kemudian dikirimkan ke basisdata. Ketika basisdata mengembalikan sekumpulan data yang berbentuk row data tersebut oleh Dlinq akan diterjemahkan menjadi objek untuk kemudian diproses pada bagian aplikasi. Ketika objek tersebut telah selesai diproses maka objek tersebut akan kembali diterjemahkan menjadi sebuah statement SQL atau stored procedure untuk disimpan di basis data.

Dasar LINQ to SQL

Sebelum mulai menggunakan LINQ to SQL ada 2 hal utama yang perlu anda tanamkan sebelum mulai menggunakannya untuk melakukan operasi terhadap basisdata. Langkah-langkah tersebut adalah :

1. Buat sebuah kelas yang memetakan kolom dari tabel anda yang ada di basisdata relasional kemudian tambahkan atribut LINQ yang sesuai.
2. Buat sebuah objek DataClasses yang akan berfungsi sebagai jembatan antara kelas yang telah anda buat dan basisdata.

Langkah-langkah diatas mungkin sekarang akan terasa sangat asing, namun mari kita sama-sama membahas langkah-langkah diatas secara mendetail pada bagian berikut ini.

Pembuatan kelas entitas

Pada langkah yang pertama ini kita akan membuat sebuah kelas entitas. Kelas entitas adalah sebuah kelas yang mewakili sebuah tabel yang ada di basis data relasional. Misalkan anda memiliki sebuah tabel Employee dengan skema seperti ini :

	Column Name	Data Type	Allow Nulls
1	ID	int	<input type="checkbox"/>
2	Name	varchar(50)	<input checked="" type="checkbox"/>
3	Address	text	<input checked="" type="checkbox"/>
4	RoleId	int	<input checked="" type="checkbox"/>

Terlihat dalam gambar diatas bahwa tabel Employee memiliki 4 buah kolom yaitu ID,Name,Address dan terakhir RoleId. Maka jika kita buat sebuah kelas yang mewakili tabel diatas hasilnya adalah sebagai berikut :

```
class Employee
{
    public int ID { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public int RoleId { get; set; }
}
```

Namun kelas diatas tidak dapat langsung kita pakai untuk dioperasikan dengan LINQ to SQL. Karena tentunya kelas diatas hanya akan dianggap sebagai kelas biasa ketika dikompilasi. Jika kita ingin menggunakan kelas diatas maka pertama-tama kita perlu memberi tahu kepada kompiler bahwa kelas ini adalah kelas yang mewakili sebuah tabel di basis data. Hal ini dapat dilakukan dengan menggunakan attribut Table yang ada pada namespace seperti dibawah ini :

```
[Table(Name="Employee")]
class Employee
```

Atribut diatas akan memberi tahu bahwa kelas ini akan merepresentasikan tabel bernama Employee. Namun sampai langkah ini kita hanya memberitahu nama tabel yang direpresentasikan. Kita belum memasangkan properti yang ada di kelas Employee dengan kolom yang ada di tabel Employee. Hal ini dapat kita lakukan juga dengan menggunakan atribut. Namun atribut yang akan kita pakai adalah atribut Column. Atribut ini memiliki beberapa properti yang dapat dilihat pada tabel dibawah ini :

Properti	Keterangan
AutoSync	Properti ini menginstruksikan common language runtime (CLR) untuk mengambil ulang nilai kolom ini didatabase setelah dilakukan operasi update atau insert
CanBeNull	Properti ini menandakan apakah kolom bisa

	mengandung nilai null
DbType	Properti yang menandakan tipe data dari kolom ini di basisdata
Expression	Properti yang mendefinisikan bagaimana perhitungan dari sebuah kolom di basisdata
IsDbGenerated	Properti yang menandakan apakah nilai data dari kolom ini dihasilkan secara otomatis oleh basis data atau tidak
IsDiscriminator	Properti ini menandakan apakah kolom ini menjadi nilai discriminator untuk hierarki LINQ to SQL
IsPrimaryKey	Properti ini menandakan apakah kolom ini menjadi primarykey di basis data
IsVersion	Properti ini menandakan apakah kolom ini merupakan tipe timestamp atau version number dalam basis data
Name	Properti ini mewakili nama kolom di basis data
UpdateCheck	Properti yang menspesifikasi bagaimana perilaku LINQ to SQL saat menghadapi konkurensi

Sehingga dengan memperhatikan properti pada tabel diatas, maka kode dari kelas Employee dapat kita modifikasi menjadi seperti berikut :

```
[Table(Name="Employee")]
class Employee
{
    [Column(Name="ID", DbType="int", IsPrimaryKey=true, IsDbGenerated=true)]
    public int ID { get; set; }
    [Column(Name = "Name", DbType = "NVarChar(50)")]
    public string Name { get; set; }
    [Column(Name = "Address", DbType = "text")]
    public string Address { get; set; }
    [Column(Name = "RoleId", DbType = "int")]
    public int RoleId { get; set; }
    public override string ToString()
    {
        return "Nama = "+Name + "\tAlamat =" +Address;
    }
}
```

Jika kita perhatikan maka seluruh properti dari kelas Employee telah dipetakan ke tabel Employee. Dalam kode diatas kita memetakan setiap properti sesuai dengan nama kolom yang diwakilinya di basis data lengkap dengan tipenya di dalam basisdata. Namun khusus untuk pemetaan kolom “ID” ke properti ID pada atributnya ditambahkan dua buah atribut yaitu IsPrimaryKey diberikan nilai true karena kolom ini menjadi primary key di basisdata dan IsDbGenerated juga diberikan nilai true karena nilai dari kolom ini bersifat autoincrement yang berarti nilainya diisi oleh basis data. Kemudian saya sengaja mengimplementasikan ulang prosedur ToString() hal ini untuk mempermudah proses penampilan hasil.

Pembuatan DataContext

Setelah membuat kelas yang memetakan kolom di basis data maka sekarang adalah saatnya kita membuat sebuah kelas turunan dari DataContext yang akan menjadi jembatan antara kelas yang telah

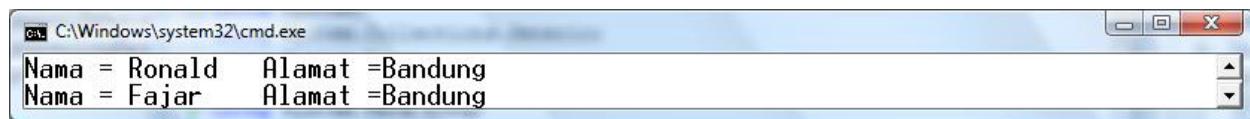
kita buat dengan kolom yang diwakilinya di basisdata. Langkahnya sangat mudah, kita hanya perlu membuat sebuah kelas yang diturunkan dari `DataContext` dan memiliki sebuah properti yang berupa sebuah objek `Table<Employee>` yang akan mewakili tabel `Employee` yang ada di basisdata. Sehingga kelas yang baru ini akan didefinisikan seperti dibawah ini :

```
class EmployeeDataContext : DataContext
{
    public Table<Employee> Employees;
    public EmployeeDataContext(string conn) : base(conn)
    {
    }
}
```

Setelah anda menyelesaikan kedua langkah diatas maka anda telah siap untuk melakukan query di program utama anda. Sekarang coba ketikkan kode dibawah ini pada program utama anda.

```
static void Main(string[] args)
{
    //ganti string koneksi sesuai dengan kebutuhan
    EmployeeDataContext data = new EmployeeDataContext(@"Data
Source=RONALDR-LAPTOP\SQLEXPRESS;Initial Catalog=LinqDatabase;Integrated
Security=True");
    var pegawai = data.Employees;
    foreach (var peg in pegawai)
        Console.WriteLine(peg);
}
```

Maka hasil dari program diatas adalah :



The screenshot shows a Windows Command Prompt window titled "cmd C:\Windows\system32\cmd.exe". The window contains the following text:
Nama = Ronald Alamat =Bandung
Nama = Fajar Alamat =Bandung

Menarik bukan? Terlihat bahwa sekarang anda dapat dengan mudah melakukan operasi pembacaan data dari basis data.

Penggunaan Tools pendukung LINQ to SQL

Dalam bagian diatas ada satu hal yang kurang, yaitu kita masih repot saat menulis kode untuk kelas yang memetakan kolom pada tabel di basis data. Coba bayangkan jika anda memiliki 20 tabel maka tentu anda harus menulis 20 kelas, cukup merepotkan bukan? Tapi jika anda menggunakan Visual Studio anda dapat menggunakan sebuah tools yang akan secara otomatis memetakan tabel anda menjadi sebuah kelas siap pakai. Atau jika anda tidak ingin menggunakan Visual Studio anda dapat menggunakan SQL metal. Jadi mari kita lihat bagaimana kedua tools ini in action.

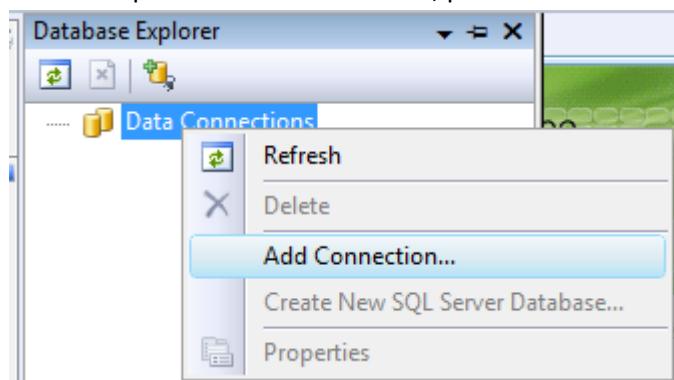
Visual Studio Object Relational Designer

Tools ini terdapat dalam Visual Studio 2008 baik dalam express, professional maupun team suite. Dengan menggunakan tools ini maka kita dapat melakukan "drag and drop" untuk melakukan design dari kelas yang akan memetakan tabel kita di basis data dan tools ini kemudian secara otomatis akan

menghasilkan kode dari kelas tersebut. Tertarik untuk menggunakan tools ini? Mari kita uji tools ini pada Hands On Lab berikut :

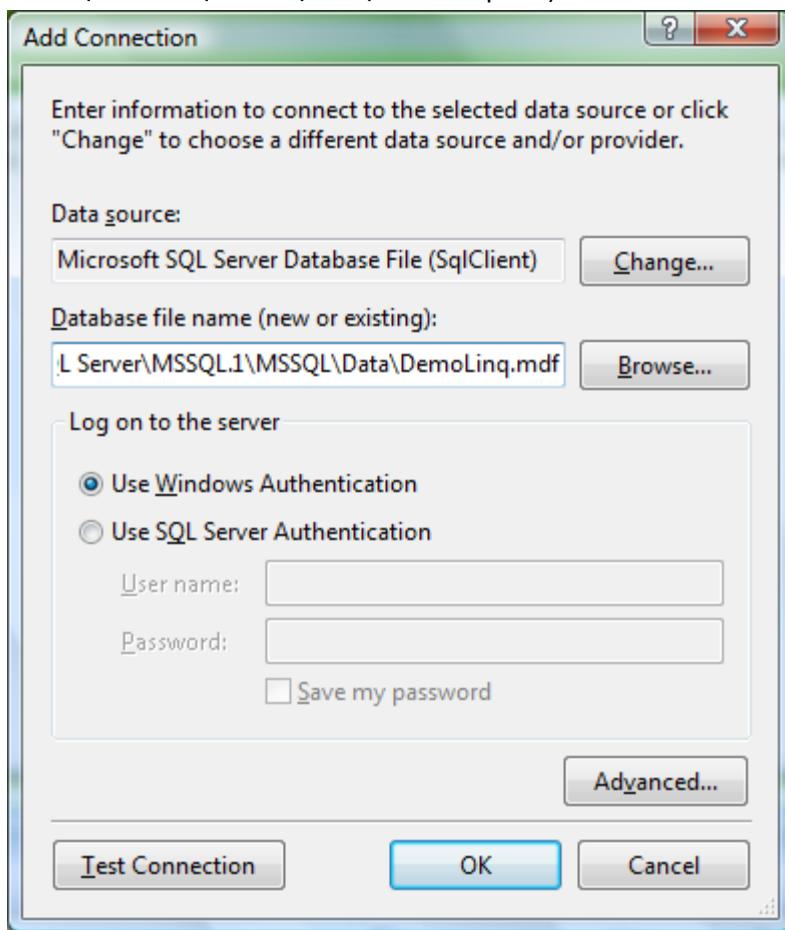
Task 1 : Menyiapkan Data Connections

1. Pastikan anda telah membuat database yang ada di [lampiran A](#)
2. Buka Visual Studio Express anda dan buat sebuah project baru dengan memilih Console Application.
3. Buka jendela “Database Explorer” hal ini bisa anda lakukan secara cepat dengan menekan kombinasi tombol Ctrl+W+L secara bersamaan.
4. Klik kanan pada “Data Connections”, pilih **Add Connection...**

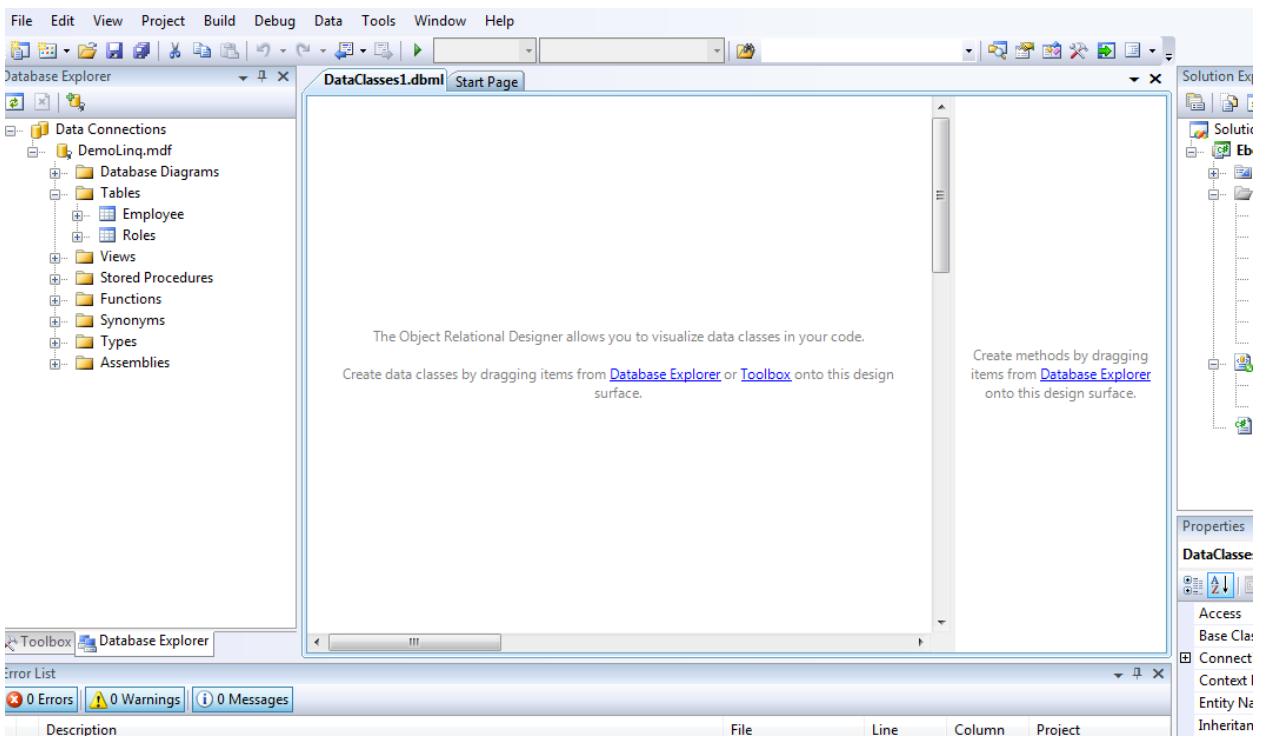


5. Pastikan pilihan Data Source adalah Microsoft SQL Server Database File (SQLClient)
6. Kemudian tekan tombol Browse dan pilih file mdf yang sesuai dengan database yang telah anda buat di [lampiran A](#) (dalam komputer penulis ada di C:\Program Files\Microsoft SQL

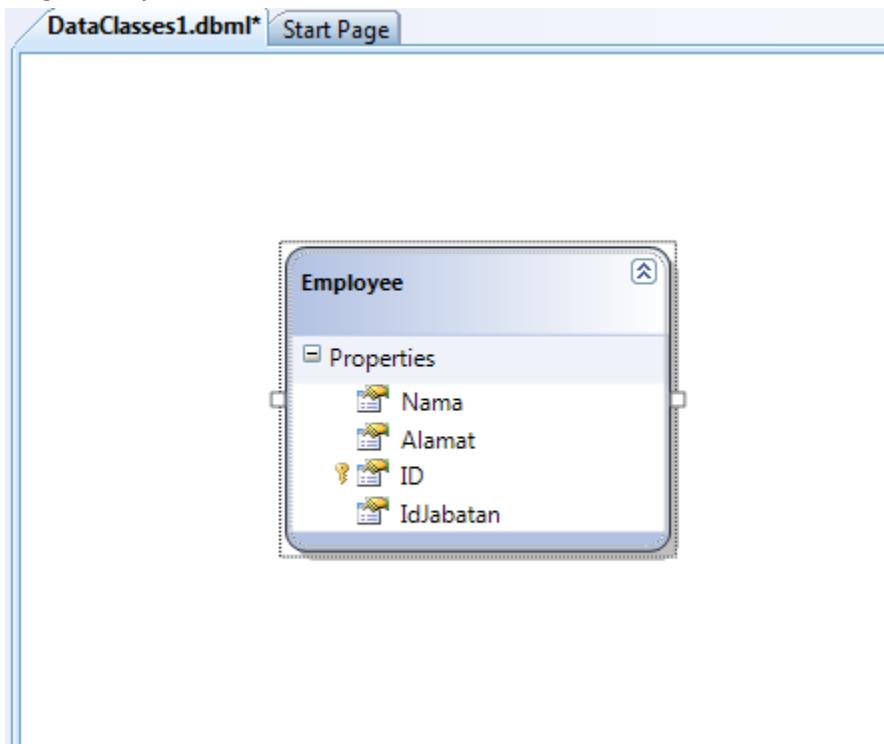
Server\MSSQL.1\MSSQL\Data\DemoLinq.mdf)



7. Klik kanan pada nama project di jendela “Solution Explorer”, pilih Add > New Item ...
8. Pilih LINQ to SQL Classes dan beri nama DataClasses.dbml lalu Add.
9. Sekarang seharusnya anda telah disajikan jendela Object Relational Designer seperti pada gambar dibawah ini :



10. Kemudian anda tinggal mendrag table Employee dari jendela Database Explorer ke bagian tengah dari jendela ORD.



11. Jika anda memperhatikan maka setelah anda mendrop table tersebut maka akan pada file "DataClasses1.designer.cs" akan dibuatkan sebuah kelas Employee yang memetakan tabel Employee di database. Selain itu akan juga dibuatkan kelas turunan DataContext yang bernama [DataClasses1DataContext](#).

12. Setelah menggunakan tools ORD, maka sekarang saatnya kita menuliskan kode untuk memperoleh data dari basisdata pada program utama seperti dibawah ini :

```
static void Main(string[] args)
{
    DataClasses1DataContext data = new DataClasses1DataContext();
    foreach (var hasil in data.Employees)
    {
        Console.WriteLine("Nama = {0} \tAlamat = {1}",
            hasil.Name,
            hasil.Address);
    }
}
```

13. Hasil dari kode diatas adalah :



```
C:\Windows\system32\cmd.exe
Nama = Ronald    Alamat = Bandung
Nama = Fajar     Alamat = Bandung
```

Terlihat dari HOL diatas bahwa dengan menggunakan ORD maka pekerjaan kita sebagai developer untuk urusan pengambilan data akan menjadi sangat mudah dengan kehadiran LINQ to SQL dan juga Visual Studio 2008.

SQL Metal

Contoh HOL pada bagian sebelumnya terlihat amat memudahkan pengembangan, namun apa yang terjadi jika kita tidak memiliki Visual Studio 2008 sebagai IDE? Tenang saja kita masih dapat menggunakan SQLMetal yang ada dalam .NET 3.5 SDK. SQL metal adalah sebuah tool berbasis command line yang berguna untuk menghasilkan sebuah kelas entitas yang memetakan sebuah tabel di basis data secara otomatis lengkap dengan properti yang terkait. Fungsionalitas utama yang ditawarkan SQLMetal sama dengan ORD namun minus tampilan GUI yang menarik.

Bagaimakah penggunaan SQLMetal ini? Mari kita lihat dalam HOL dibawah ini :

Task 1 : penggunaan SQL Metal

HOL ini akan berhubungan dengan HOL sebelumnya. Karena pada intinya HOL ini hanya akan mengganti kelas DataContext yang sebelumnya telah dihasilkan dari penggunaan ORD.

1. Buka Command Prompt.
2. Arahkan Command Prompt anda ke “*drive:\Program Files\Microsoft SDKs\Windows\v*n.nn\bin**.”
n.nn adalah versi dari sistem operasi anda. Sehingga jika anda menggunakan Vista *v*n.nn** akan diganti menjadi v6.0A.
3. Ketikkan perintah “*sqlmetal /server:ronaldr-laptop\sqlexpress /database:LinqDatabase /pluralize /code:d:/dataContext.cs*” tekan enter.

Catatan :

1. Sesuaikan nama server dengan nama komputer anda. Nama komputer penulis adalah ronaldr-laptop sehingga ganti nama ini dengan nama komputer anda.
2. Sesuaikan lokasi file hasil. Pada contoh diatas SQLMetal akan menuliskan file hasil pada drive D.
4. Setelah perintah diatas dieksekusi maka secara otomatis akan dihasilkan sebuah file dengan nama "dataContext" pada lokasi yang anda tentukan.
5. Jika anda buka file dataContext.cs maka file tersebut akan mirip dengan file "DataClasses1.designer.cs" yang telah kita hasilkan pada HOL sebelumnya.
6. Buka kembali project yang telah anda buat pada HOL sebelumnya. Sekarang kita akan menguji apakah kelas entitas yang telah dihasilkan oleh SQLMetal tersebut dapat digunakan atau tidak.
7. Klik kanan pada nama project, pilih **Add** kemudian **Existing Item ...** lalu tunjuk lokasi file "dataContext.cs" yang telah dibuat pada langkah 3.



8. Ganti kode pada prosedur Main anda menjadi seperti dibawah ini :

```

LinqDatabase data = new LinqDatabase(@"Data Source=ronaldr-
laptop\sqlexpress;Initial Catalog=LinqDatabase;Integrated Security=True");
foreach (var hasil in data.Employees)
{
    Console.WriteLine("Nama = {0} \tAlamat = {1}",
                      hasil.Name,
                      hasil.Address);
}

```

Catatan :

1. Ganti string koneksi sesuai dengan komputer anda. Pada umumnya anda hanya tinggal mengganti nama "ronaldr-laptop" menjadi nama komputer anda.

9. Tekan F5 untuk menjalankan kode diatas. Maka hasil keluaran dari program ini akan sama seperti pada HOL sebelumnya.

Terlihat dalam HOL diatas bahwa penggunaan SQLMetal dapat menggantikan ORD jika anda tidak ingin menggunakan Visual Studio untuk menghasilkan kelas entitas anda. Namun argumen untuk penggunaan SQLMetal masih ada lebih banyak lagi dibanding yang telah kita gunakan pada HOL. Daftar argumen secara lengkap dapat dilihat pada tabel dibawah ini :

Argumen	Keterangan
/server : <name>	Argumen menspesifikasikan nama server basis data yang akan anda gunakan. Jika anda tidak menuliskan argumen ini maka SQLMetal akan menggunakan localhost/sqlexpress
/database: <name>	Argumen ini menspesifikasikan nama database yang akan digunakan untuk menghasilkan kelas entitas
/user : <name>	Argumen ini mewakili nama pengguna yang digunakan untuk otentifikasi saat koneksi ke basis data
/password : <name>	Argumen ini mewakili password pengguna yang digunakan untuk otentifikasi saat koneksi ke basis data
/views	Argumen ini digunakan apabila anda ingin agar SQLMetal membuat properti Table<T> dan kelas entitas untuk mendukung views yang ada dalam database
/functions	Argumen ini digunakan agar SQLMetal menghasilkan fungsi yang akan memanggil fungsi yang dibuat oleh pengguna pada basis data
/sprocs	Argumen ini digunakan agar SQLMetal menghasilkan fungsi yang mewakili stored procedure yang ada di basis data
/dbml [:file]	Argumen ini digunakan untuk menspesifikasikan nama file DBML. Tujuan dari pembuatan file ini adalah agar kita dapat mengendalikan nama kelas dan properti dari kelas entitas yang akan dihasilkan.
/code [:file]	Argumen ini digunakan untuk menspesifikasikan nama file yang akan dihasilkan oleh SQLMetal.
/map [:file]	Argumen ini digunakan untuk menspesifikasikan sebuah file <i>mapping</i> dengan format XML. Sehingga dengan dihasilkannya file ini maka kita dapat menggunakan LINQ to SQL tanpa source code LINQ to SQL kita ikut dikompilasi
/language : <language>	Argumen ini digunakan untuk menspesifikasikan

	bahasa yang akan digunakan pada kode kelas entitas kita. Opsi yang valid adalah csharp,C# dan VB
/namespace: <name>	Argumen ini digunakan untuk menspesifikasikan namespace yang akan mengandung kelas entitas.
/context:<type>	Argumen ini digunakan untuk menspesifikasikan nama dari kelas datacontext yang dihasilkan.
/entitybase :<type>	Argumen ini digunakan untuk menspesifikasikan base class dari semua kelas entitas yang dihasilkan
/pluralize	Argumen ini akan menyebabkan SQLMetal untuk membuat nama tabel menjadi singular dan plural, sebagai contoh untuk nama kelas akan singular misal Customer (singular) sedangkan nama tabel akan plural semisal Customers(plural)
/serialization : <option>	Argumen ini digunakan untuk menspesifikasikan apakah SQLMetal akan menghasilkan atribut serialization pada kelas yang dihasilkan

Operasi terhadap basis data dengan LINQ to SQL

Jika pada perjalanan sebelumnya kita telah bersama-sama melihat bagaimana kita dapat melakukan pemetaan dari basis data ke kelas entitas maka pada pemberhentian kali ini kita akan mulai melakukan eksplorasi terhadap LINQ to SQL. Pada bagian ini kita akan melihat bagaimana penggunaan LINQ to SQL untuk melakukan operasi berikut pada basis data :

1. Insert (Pemasukan data baru)
2. Query (Pengambilan data dari basis data)
3. Update (Pengubahan data)
4. Delete (Penghapusan Data)

Namun untuk mulai melakukan perjalanan dalam bagian ini anda perlu menyiapkan sebuah project yang akan kita gunakan dalam proses explorasi ini. project yang akan dibuat ini dapat anda lihat pada [lampiran A](#). Selama bagian ini saya usahakan bahwa anda hanya tinggal mengubah kode prosedur main untuk setiap contoh kasus.

Pemasukan data dengan menggunakan LINQ to SQL

Pada bagian ini kita akan sama-sama melihat bagaimana operasi penambahan data dilakukan dengan menggunakan LINQ to SQL. Pada dasarnya operasi penambahan data dilakukan dengan empat langkah yaitu :

1. Buat sebuah objek dari kelas DataContext yang sesuai dengan database yang akan kita jadikan target operasi.
2. Buat sebuah objek dari kelas entitas yang bersesuaian dengan data yang akan kita tambahkan.
3. Tambajkan objek yang dihasilkan pada langkah kedua kedalam koleksi tabel yang ada dalam objek datacontext yang ada di langkah pertama.

- Lakukan pemanggilan prosedur SubmitChanges pada objek DataContext. Hal ini dilakukan untuk menyimpan perubahan data ke database. Jika anda tidak melakukan hal ini maka semua perubahan yang dilakukan tidak akan disimpan ke database.

Langkah yang cukup sederhana, apalagi jika kita mengingat bagaimana rumitnya operasi penambahan data jika menggunakan ADO .NET. Runtutan keempat langkah diatas dapat dilihat pada kode dibawah ini :

```
static void Main(string[] args)
{
    // 1. pembuatan objek datacontext
    LINQDataClassesDataContext data = new LINQDataClassesDataContext();
    // 2. pembuatan objek dari kelas entitas yang bersesuaian
    Employee pegawai = new Employee() {
        Name = "Dwi", RoleID = 1, Address = "Jakarta" };
    // 3. penambahan objek kepada tabel
    data.Employees.InsertOnSubmit(pegawai);
    // 4. menyimpan perubahan ke database
    data.SubmitChanges();
}
```

Jika anda kembali melihat data yang ada di basis data maka anda akan melihat bahwa telah ada sebuah row baru di basis data yang berisikan data sesuai dengan objek Employee yang anda masukkan. Kemudian jika anda ingin melihat query SQL yang telah dilakukan oleh LINQ to SQL maka anda dapat menambahkan baris berikut tepat setelah pembuatan objek DataContext:

```
data.Log = Console.Out;
```

Maka dilayar akan ditampilkan hasil seperti pada gambar dibawah. Terlihat bahwa dilayar ditampilkan query SQL yang dilakukan terhadap basis data.

```
C:\Windows\system32\cmd.exe
INSERT INTO [dbo].[Employee]([Name], [Address], [RoleID])
VALUES (@p0, @p1, @p2)
SELECT CONVERT(Int,SCOPE_IDENTITY()) AS [value]
-- @p0: Input NVarChar (Size = 3; Prec = 0; Scale = 0) [Dwi]
-- @p1: Input Text (Size = 7; Prec = 0; Scale = 0) [Jakarta]
-- @p2: Input Int (Size = 0; Prec = 0; Scale = 0) [1]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8
```

Salah satu hal menarik dari proses pemasukan data dengan menggunakan LINQ to SQL ini adalah kemampuan dari DataContext yang dihasilkan dari ORD atau SQLMetal untuk mendeteksi hubungan antar objek dari kelas entitas yang saling berhubungan. Sehingga jika misalnya salah satu dari objek tersebut disimpan maka objek yang behubungannya pun akan turut disimpan. Contohnya pada kode dibawah ini :

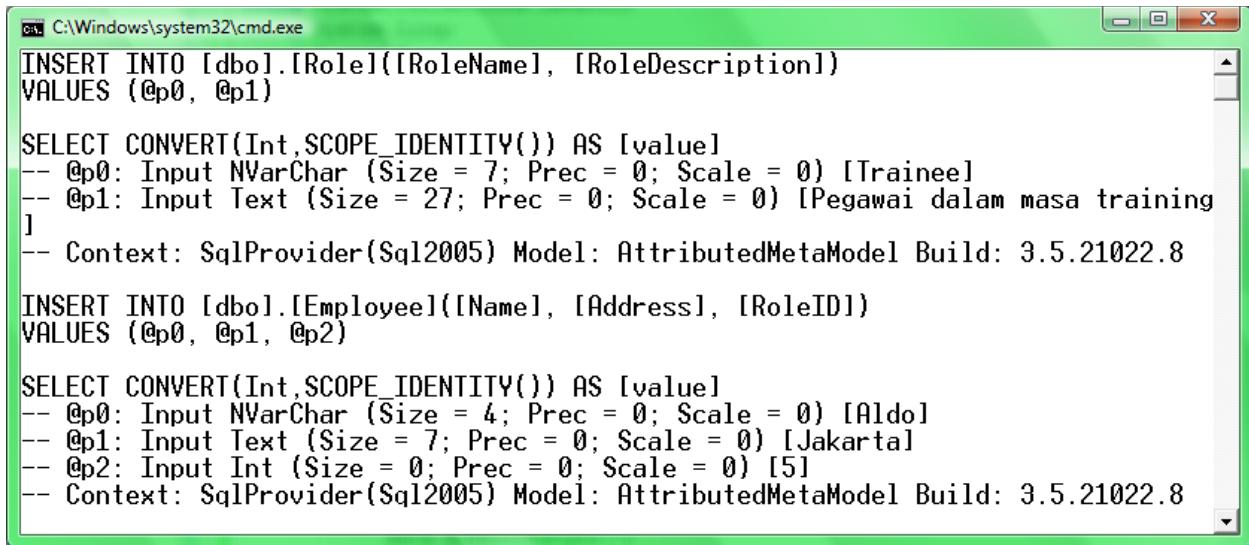
```
LINQDataClassesDataContext data = new LINQDataClassesDataContext();
data.Log = Console.Out;
Employee pegawai = new Employee() {
```

```

        Name = "Aldo", Address = "Jakarta" };
Role jabatan = new Role()
{
    RoleDescription = "Pegawai dalam masa training",
    RoleName = "Trainee"
};
pelanggan.Role = jabatan;
data.Employees.InsertOnSubmit(pegawai);
data.SubmitChanges();

```

Pada kode diatas kita menambahkan sebuah pegawai(employee) baru beserta dengan sebuah jabatan(Role) baru. Namun kita hanya menyimpan objek pegawai saja ke dalam tabel Employees yang ada pada DataContext. Tetapi DataContext secara cerdas akan mendeteksi bahwa objek yang baru kita tambahkan tersebut memiliki keterhubungan dengan objek jabatan yang belum tersimpan dalam basis data, sehingga objek jabatan tersebut pun akan turut disimpan. Hal ini terlihat dari Query SQL yang tertulis pada hasil program ini. terlihat jelas bahwa sebelum menyimpan data employee terlebih dahulu disimpan data role.



```

C:\Windows\system32\cmd.exe
INSERT INTO [dbo].[Role]([RoleName], [RoleDescription])
VALUES (@p0, @p1)

SELECT CONVERT(Int,SCOPE_IDENTITY()) AS [value]
-- @p0: Input NVarChar (Size = 7; Prec = 0; Scale = 0) [Trainee]
-- @p1: Input Text (Size = 27; Prec = 0; Scale = 0) [Pegawai dalam masa training]
] -- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

INSERT INTO [dbo].[Employee]([Name], [Address], [RoleID])
VALUES (@p0, @p1, @p2)

SELECT CONVERT(Int,SCOPE_IDENTITY()) AS [value]
-- @p0: Input NVarChar (Size = 4; Prec = 0; Scale = 0) [Aldo]
-- @p1: Input Text (Size = 7; Prec = 0; Scale = 0) [Jakarta]
-- @p2: Input Int (Size = 0; Prec = 0; Scale = 0) [5]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

```

Melakukan permintaan data dengan menggunakan LINQ to SQL (Query)

Jika pada bagian sebelumnya kita telah melihat bagaimana kita dapat melakukan permintaan data (Query) dengan menggunakan LINQ to SQL. Query pada LINQ to SQL sama dengan query pada LINQ seperti umumnya sehingga tentunya semua SQO dapat juga kita gunakan.

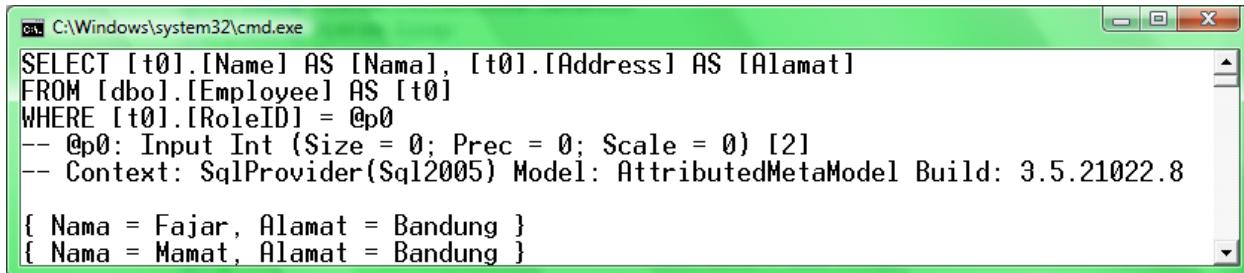
Contoh penggunaan Query pada LINQ to SQL ada pada kode dibawah ini:

```

LINQDataClassesDataContext data = new LINQDataClassesDataContext();
data.Log = Console.Out;
var hasil = from pegawai in data.Employees
           where pegawai.RoleID == 2
           select new { Nama = pegawai.Name, Alamat =
pegawai.Address };
foreach (var pegawai in hasil)
    Console.WriteLine(pegawai);

```

Kode diatas dapat kita lihat akan melakukan query sederhana untuk memperoleh data dari Employee yang ada dalam basis data. Sehingga bentuk querynya pun akan sederhana seperti yang dapat kita lihat pada hasil dibawah ini :

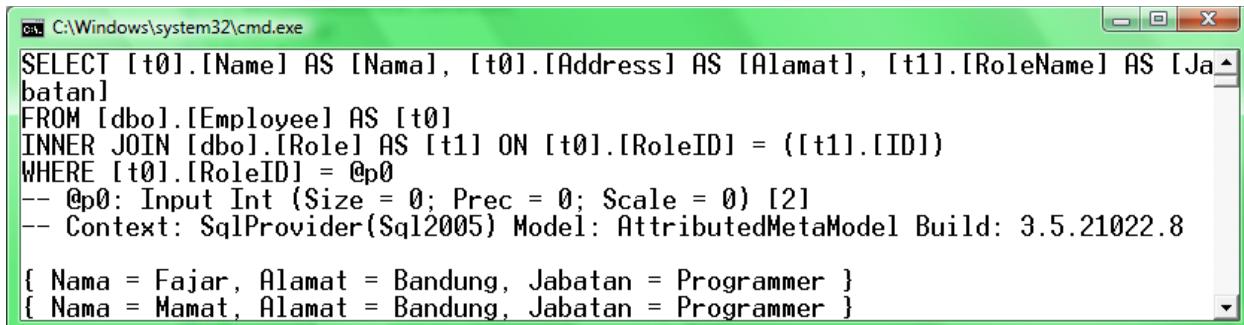


```
C:\Windows\system32\cmd.exe
SELECT [t0].[Name] AS [Nama], [t0].[Address] AS [Alamat]
FROM [dbo].[Employee] AS [t0]
WHERE [t0].[RoleID] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [2]
-- Context: SqlProvider(Sql12005) Model: AttributedMetaModel Build: 3.5.21022.8
{ Nama = Fajar, Alamat = Bandung }
{ Nama = Mamat, Alamat = Bandung }
```

Seperti yang telah disinggung pada awal bagian ini, maka kita dapat menggunakan SQO dalam LINQ to SQL, sehingga dengan menggunakan SQO kita dapat melakukan operasi join dengan mudah pada query kita seperti yang dicontohkan pada kode dibawah ini:

```
LINQDataClassesDataContext data = new LINQDataClassesDataContext();
data.Log = Console.Out;
var hasil = from pegawai in data.Employees
join jabatan in data.Roles
on pegawai.RoleID equals jabatan.ID
where pegawai.RoleID == 2
select new { Nama = pegawai.Name, Alamat =
pegawai.Address, Jabatan=jabatan.RoleName };
foreach (var pegawai in hasil)
Console.WriteLine(pegawai);
```

Dalam kode diatas terlihat bahwa kita melakukan operasi join dengan menggunakan SQO yang kemudian diterjemahkan menjadi query SQL seperti pada gambar dibawah ini.



```
C:\Windows\system32\cmd.exe
SELECT [t0].[Name] AS [Nama], [t0].[Address] AS [Alamat], [t1].[RoleName] AS [Jabatan]
FROM [dbo].[Employee] AS [t0]
INNER JOIN [dbo].[Role] AS [t1] ON [t0].[RoleID] = ([t1].[ID])
WHERE [t0].[RoleID] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [2]
-- Context: SqlProvider(Sql12005) Model: AttributedMetaModel Build: 3.5.21022.8
{ Nama = Fajar, Alamat = Bandung, Jabatan = Programmer }
{ Nama = Mamat, Alamat = Bandung, Jabatan = Programmer }
```

Terlihat bahwa LINQ to SQL akan menghasilkan sebuah query yang melakukan operasi inner join pada tabel Employee dengan Role. Namun kita pun dapat tidak menggunakan SQO untuk mendapatkan hasil join seperti diatas. Kode dibawah ini akan menghasilkan hasil yang sama dengan kode sebelumnya :

```
LINQDataClassesDataContext data = new LINQDataClassesDataContext();
data.Log = Console.Out;
var hasil = from pegawai in data.Employees
where pegawai.RoleID == 2
```

```

        select new { Nama = pegawai.Name, Alamat =
pegawai.Address, Jabatan=pegawai.Role.RoleName };
    foreach (var pegawai in hasil)
        Console.WriteLine(pegawai);

```

Hasil dari kode diatas akan sama dengan kode sebelumnya. Karena LINQ to SQL telah mengetahui adanya hubungan antara tabel pegawai dengan tabel role. Hal ini mudah dilihat karena jika anda melihat dengan seksama properti kelas Employee maka akan ada properti Role yang akan menunjuk ke objek Role yang sesuai. Sehingga operasi join akan secara otomatis dilakukan ketika anda memakai value dari properti Role dari objek Employee seperti pada kode diatas. Namun ada sedikit perbedaan dengan menggunakan operasi join ialah jika menggunakan Join pada SQO akan diterjemahkan sebagai inner join namun pada kode ini akan diterjemahkan menjadi left outer join seperti pada gambar dibawah ini :

```

C:\Windows\system32\cmd.exe
SELECT [t0].[Name] AS [Nama], [t0].[Address] AS [Alamat], [t1].[RoleName] AS [Jabatan]
FROM [dbo].[Employee] AS [t0]
LEFT OUTER JOIN [dbo].[Role] AS [t1] ON [t1].[ID] = [t0].[RoleID]
WHERE [t0].[RoleID] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [2]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8
{ Nama = Fajar, Alamat = Bandung, Jabatan = Programmer }
{ Nama = Mamat, Alamat = Bandung, Jabatan = Programmer }

```

Hal ini wajar mengingat semua properti pada objek Employee perlu diisi meskipun objek tersebut tidak memiliki padanan pada tabel role.

Melakukan pengubahan data (Update) dengan menggunakan LINQ to SQL

Melakukan perubahan data pada LINQ to SQL cukup sederhana. Operasi ini dapat dilakukan semudah mengubah properti dari sebuah objek dari kelas entitas dan kemudian memanggil prosedur SubmitChanges pada objek DataContext yang sesuai. Kemudian menangani isu konkurensi yang mungkin timbul. Isu konkurensi ini akan coba kita explorasi pada bagian akhir dari LINQ to SQL.

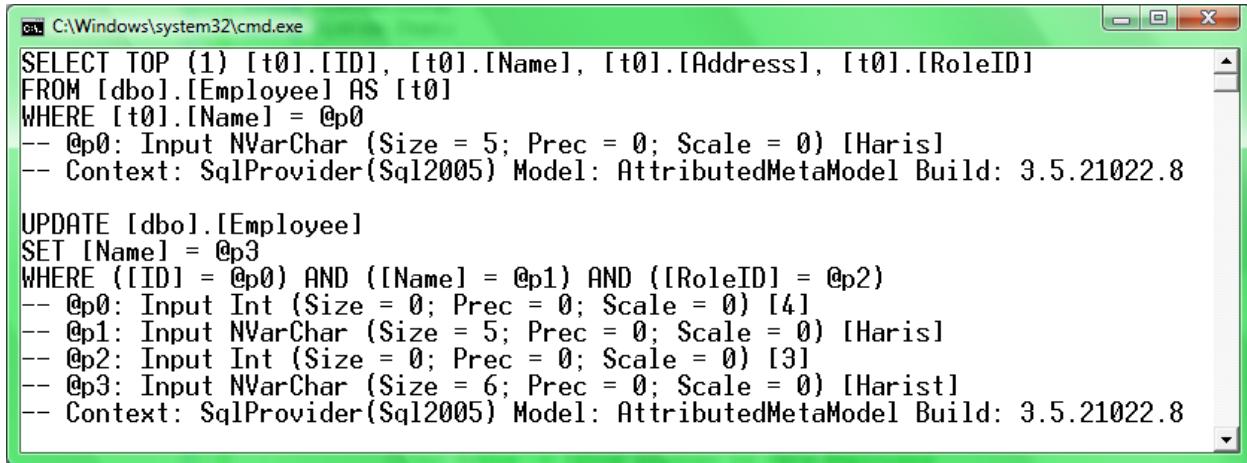
Contoh perubahan data dapat dilihat pada kode dibawah ini :

```

LINQDataClassesDataContext data = new LINQDataClassesDataContext();
data.Log = Console.Out;
var Haris = (from pegawai in data.Employees
            where pegawai.Name.Equals("Haris")
            select pegawai).First();
Haris.Name = "Harist";
data.SubmitChanges();

```

Kode diatas akan mengubah nama dari pegawai yang bernama Haris menjadi Harist. Jika anda melihat log query SQL yang akan ditampilkan oleh program ini maka terlihat bahwa telah dilakukan operasi update :



```
C:\Windows\system32\cmd.exe
SELECT TOP (1) [t0].[ID], [t0].[Name], [t0].[Address], [t0].[RoleID]
FROM [dbo].[Employee] AS [t0]
WHERE [t0].[Name] = @p0
-- @p0: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [Haris]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

UPDATE [dbo].[Employee]
SET [Name] = @p3
WHERE ([ID] = @p0) AND ([Name] = @p1) AND ([RoleID] = @p2)
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [4]
-- @p1: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [Haris]
-- @p2: Input Int (Size = 0; Prec = 0; Scale = 0) [3]
-- @p3: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [Harist]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8
```

Namun yang menarik untuk operasi update dengan menggunakan LINQ to SQL adalah adanya kemampuan dari `DataContext` yang dihasilkan dari `ORD` atau `SQLMetal` untuk mendeteksi adanya perubahan hubungan relasi di antara objek dan merefleksikan perubahannya di basis data. Contoh singkatnya ialah pada objek `Employee` dan `Role` yang kita gunakan. Kedua buah objek tersebut memiliki keterhubungan foreign key di dalam basis data. Sehingga kita dapat menganggap bahwa `Employee` adalah `Child` dari `Role`. Sehingga perubahan keterhubungan antara keduanya dapat dilakukan dengan mengganti referensi elemen `Role` dari `Employee` (`Child`) atau membuang/menambahkan objek `Employee` ke dalam koleksi `Employees` dari `Role` (`Parent`).

Contoh kode dibawah ini akan mengubah properti `RoleID` dari sebuah objek `Employee` dengan cara menganti referensi elemen `Role` dari objek tersebut.

```
LINQDataClassesDataContext data =
    new LINQDataClassesDataContext();
data.Log = Console.Out;
//memperoleh objek Employee yang sesuai
var PegawaiPertama = (from pegawai in data.Employees
                       where pegawai.ID == 1
                       select pegawai).First();
//memperoleh objek Role yang sesuai
var RoleProgrammer = (from jabatan in data.Roles
                       where jabatan.ID == 2
                       select jabatan).First();
//mengubah Role dari Employee
PegawaiPertama.Role = RoleProgrammer;
data.SubmitChanges();
```

Hasil dari kode diatas adalah:

```
C:\Windows\system32\cmd.exe
SELECT TOP (1) [t0].[ID], [t0].[Name], [t0].[Address], [t0].[RoleID]
FROM [dbo].[Employee] AS [t0]
WHERE [t0].[ID] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [1]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

SELECT TOP (1) [t0].[ID], [t0].[RoleName], [t0].[RoleDescription]
FROM [dbo].[Role] AS [t0]
WHERE [t0].[ID] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [2]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

SELECT [t0].[ID], [t0].[RoleName], [t0].[RoleDescription]
FROM [dbo].[Role] AS [t0]
WHERE [t0].[ID] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [1]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

UPDATE [dbo].[Employee]
SET [RoleID] = @p3
WHERE ([ID] = @p0) AND ([Name] = @p1) AND ([RoleID] = @p2)
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [1]
-- @p1: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [Ronald]
-- @p2: Input Int (Size = 0; Prec = 0; Scale = 0) [1]
-- @p3: Input Int (Size = 0; Prec = 0; Scale = 0) [2]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8
```

Dalam log Query SQL diatas terlihat jelas bahwa perubahan referensi objek Role pada objek Employee ikut direfleksikan di basis data dengan adanya perubahan RoleID pada row Employee yang sesuai. Menarik bukan? meskipun kita tidak secara explisit merubah atribut RoleID yang ada pada objek Employee kita memperoleh perubahan RoleID pada row Employee di basis data. Dengan hal ini maka LINQ to SQL telah menunjukkan kemampuannya untuk menjembatani dunia objek dan dunia basis data relasional.

Jika pada contoh diatas kita melakukan perubahan dengan mengganti referensi pada element child maka sekarang kita akan melakukan perubahan dengan efek yang sama namun dengan cara membuang sebuah objek dari sebuah koleksi di sebuah Role dan menambahkan objek yang dibuang tadi ke Role yang lainnya. Kodennya adalah seperti berikut :

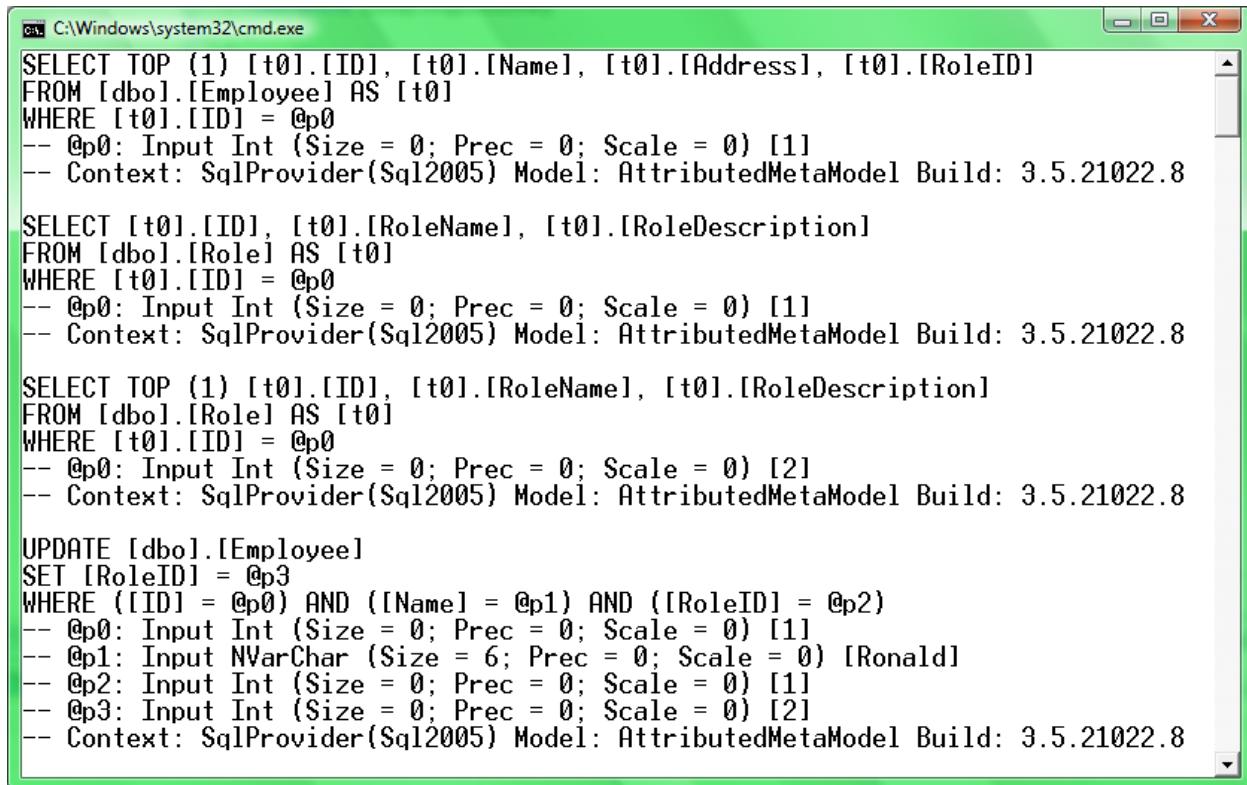
```
LINQDataClassesDataContext data =
    new LINQDataClassesDataContext();
data.Log = Console.Out;
Employee PegawaiPertama = (from pegawai in data.Employees
                            where pegawai.ID == 1
                            select pegawai).First();
//memperoleh objek Role yang sekarang menjadi parent dari objek
PegawaiPertama
var FirstRole = PegawaiPertama.Role;
//memperoleh objek Role yang baru yang akan menjadi parent dari
objek PegawaiPertama
var SecondRole = (from jabatanBaru in data.Roles
                  where jabatanBaru.ID == 2
                  select jabatanBaru).First();
```

```

        //membuang objek PegawaiPertama dari koleksi Employees objek Role
yang lama
        FirstRole.Employees.Remove(PegawaiPertama);
        //menambahkan objek PegawaiPertama ke koleksi Employees objek
Role yang baru
        SecondRole.Employees.Add(PegawaiPertama);
        data.SubmitChanges();

```

Hasil dari kode ini adalah:



```

C:\Windows\system32\cmd.exe
SELECT TOP (1) [t0].[ID], [t0].[Name], [t0].[Address], [t0].[RoleID]
FROM [dbo].[Employee] AS [t0]
WHERE [t0].[ID] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [1]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

SELECT [t0].[ID], [t0].[RoleName], [t0].[RoleDescription]
FROM [dbo].[Role] AS [t0]
WHERE [t0].[ID] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [1]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

SELECT TOP (1) [t0].[ID], [t0].[RoleName], [t0].[RoleDescription]
FROM [dbo].[Role] AS [t0]
WHERE [t0].[ID] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [2]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

UPDATE [dbo].[Employee]
SET [RoleID] = @p3
WHERE ([ID] = @p0) AND ([Name] = @p1) AND ([RoleID] = @p2)
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [1]
-- @p1: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [Ronald]
-- @p2: Input Int (Size = 0; Prec = 0; Scale = 0) [1]
-- @p3: Input Int (Size = 0; Prec = 0; Scale = 0) [2]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

```

Sama dengan hasil pada kode sebelumnya bukan?

Penghapusan Data (delete)dengan menggunakan LINQ to SQL

Operasi penghapusan data pada LINQ to SQL cukup mudah dilakukan. Pada intinya kita hanya perlu melewatkkan objek yang ingin kita hapus pada sebagai parameter dari fungsi `DeleteOnSubmit()` pada objek `Table` yang mengandung objek yang ingin kita hapus tersebut. Jika kita ingin menghapus sekumpulan objek yang ada dalam sebuah koleksi maka kita dapat melewatkkan koleksi tersebut sebagai parameter dari fungsi `DeleteAllOnSubmit()` pada objek `Table` yang mengandung objek yang ingin kita hapus tersebut.

Contoh kode dari penggunaan prosedur `DeleteOnSubmit()` adalah seperti dibawah ini :

```

LINQDataClassesDataContext data =
    new LINQDataClassesDataContext();
data.Log = Console.Out;
Employee PegawaiDihapus = (from pegawai in data.Employees

```

```

        where pegawai.Name.Equals("Mamat")
        select pegawai).First();
    data.Employees.DeleteOnSubmit(PegawaiDihapus);
    data.SubmitChanges();

```

kode diatas akan menghapus sebuah row dalam basis data yang memiliki nilai Name sama dengan "Ronald". Sehingga jika kita lihat dalam log Query SQL yang dihasilkan ada perintah DELETE seperti yang dapat dilihat pada hasil program diatas :

```

C:\Windows\system32\cmd.exe
SELECT TOP (1) [t0].[ID], [t0].[Name], [t0].[Address], [t0].[RoleID]
FROM [dbo].[Employee] AS [t0]
WHERE [t0].[Name] = @p0
-- @p0: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [Mamat]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

DELETE FROM [dbo].[Employee] WHERE ([ID] = @p0) AND ([Name] = @p1) AND ([RoleID]
= @p2)
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [5]
-- @p1: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [Mamat]
-- @p2: Input Int (Size = 0; Prec = 0; Scale = 0) [2]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

```

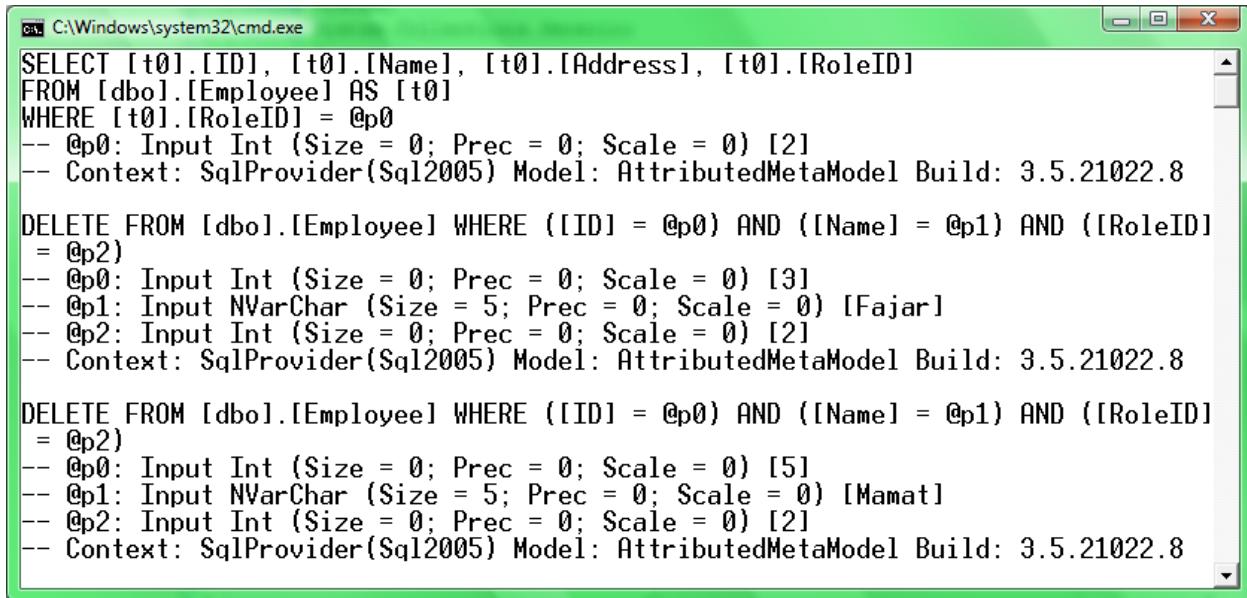
Untuk penggunaan prosedur DeleteAllOnSubmit() pun relatif sama pemakaiannya, perbedaanya hanya pada parameter masukan yang berupa koleksi seperti dibawah ini:

```

LINQDataClassesDataContext          data      =
LINQDataClassesDataContext();       = new
                                   
data.Log = Console.Out;
var PegawaiDihapus = from pegawai in data.Employees
                      where pegawai.RoleID == 2
                      select pegawai;
data.Employees.DeleteAllOnSubmit(PegawaiDihapus);
data.SubmitChanges();

```

Hasil dari kode diatas dapat dilihat pada gambar dibawah ini. Terlihat pada gambar dibawah bahwa dilakukan dua kali operasi penghapusan pada tabel Employee. Hal ini tentunya karena jumlah koleksi pada objek Pegawai Dihapus ada dua buah.



```
C:\Windows\system32\cmd.exe
SELECT [t0].[ID], [t0].[Name], [t0].[Address], [t0].[RoleID]
FROM [dbo].[Employee] AS [t0]
WHERE [t0].[RoleID] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [2]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

DELETE FROM [dbo].[Employee] WHERE ([ID] = @p0) AND ([Name] = @p1) AND ([RoleID] = @p2)
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [3]
-- @p1: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [Fajar]
-- @p2: Input Int (Size = 0; Prec = 0; Scale = 0) [2]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

DELETE FROM [dbo].[Employee] WHERE ([ID] = @p0) AND ([Name] = @p1) AND ([RoleID] = @p2)
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [5]
-- @p1: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [Mamat]
-- @p2: Input Int (Size = 0; Prec = 0; Scale = 0) [2]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8
```

Khusus untuk operasi penghapusan (Delete) ada satu hal yang perlu diingat bahwa tidak seperti operasi insert dimana objek yang terkait dengan objek yang dimasukkan akan ikut dimasukkan ke basis data, untuk operasi Delete hanya objek yang dihapus saja yang akan akan dihapus dari basis data. Sehingga objek-objek lain yang terkait dengan objek tersebut tidak akan dihapus. Sebagai contoh untuk kasus yang digunakan dalam bab ini ketika kita menghapus sebuah Role maka objek-objek Employee yang terkait dengan Role tersebut tidak akan dihapus. Kita harus menghapusnya secara implisit sebelum menghapus objek Role.

Isu Konkurensi

Isu Konkurensi merupakan sebuah isu yang sangat penting dalam dunia pengembangan perangkat lunak yang berhubungan dengan basis data. Isu ini biasanya timbul ketika ada lebih dari satu pengguna mengakses sebuah tabel di basis data dalam satu waktu yang sama. Contohnya dari isu ini adalah misalnya aplikasi kita mengambil data dari basis data dengan menggunakan LINQ, kemudian setelah aplikasi tersebut mengambil data maka ada aplikasi lain yang merubah data tersebut. Maka ketika aplikasi kita melakukan perubahan data dan mengirimkan perubahan tersebut ke basis data kita akan memperoleh sebuah exception. Contohnya ada di HOL dibawah ini :

HOL : Contoh isu konkurensi

1. Siapkan project yang ada di [lampiran A](#)
2. Ketikkan kode dibawah ini pada bagian Main di file program.cs

```
LINQDataClassesDataContext data = new LINQDataClassesDataContext();
Employee diganti = (from pegawai in data.Employees
                     where pegawai.ID == 1
                     select pegawai).First();
Console.WriteLine("Masukkan nama yang baru : ");
string nama = Console.ReadLine();
diganti.Name = nama;
try
```

```

        {
            data.SubmitChanges();
        }
        catch (ChangeConflictException e)
        {
            Console.WriteLine(e.StackTrace);
        }
    }
}

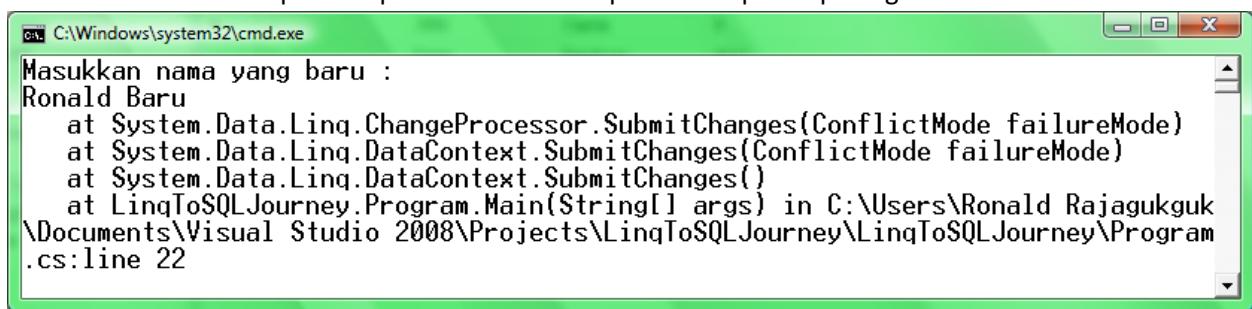
```

3. Tekan Ctrl+ F5 untuk men-debug aplikasi ini, dan biarkan terlebih dahulu.
4. Buka SQL Management Studio Express 2005.
5. Buka database LINQtoSQLJourney.
6. Buka tabel “Employee”
7. Ubah data “Ronald” menjadi “Ronald2”.

Table - dbo.Employee

	ID	Name	Address	RoleID
1	1	Ronald2	Bandung	1
2	2	Joko	Ciamis	4
3	3	Fajar	Bandung	NULL
4	4	Haris	Jakarta	3
5	5	Mamat	Bandung	NULL
*	NULL	NULL	NULL	NULL

8. Kembali ke jendela aplikasi console yang anda baru buat dan ketikkan sembarang nama.
9. Voila....anda akan memperoleh pesan kesalahan seperti ditampilkan pada gambar dibawah ini :



```

C:\Windows\system32\cmd.exe
Masukkan nama yang baru :
Ronald Baru
   at System.Data.Linq.ChangeProcessor.SubmitChanges(ConflictMode failureMode)
   at System.Data.Linq.DataContext.SubmitChanges(ConflictMode failureMode)
   at System.Data.Linq.DataContext.SubmitChanges()
   at LinqToSQLJourney.Program.Main(String[] args) in C:\Users\Ronald Rajagukguk
\Documents\Visual Studio 2008\Projects\LinqToSQLJourney\LinqToSQLJourney\Program
.cs:line 22

```

HOL sederhana diatas menunjukkan terjadinya isu konkurensi dan mari kita bedah sedikit apa yang terjadi dalam HOL diatas. Pertama-tama aplikasi dengan menggunakan LINQ to SQL akan mengambil data dari basis data. Kemudian setelah aplikasi mengambil data, aplikasi akan menunggu input masukan dari user, dan ketika sedang menunggu input masukan maka kita mengubah data di basis data. Sehingga ketika aplikasi memanggil SubmitChanges() pada DataContext maka DataContext akan mendeteksi bahwa ternyata di basis data telah terjadi perubahan data yang terjadi setelah aplikasi mengambil data, sehingga karena LINQ to SQL secara standar menggunakan Optimistic Concurrency maka akan dilempar sebuah exception.

Untuk menangani kasus diatas kita dapat menggunakan beberapa pendekatan tergantung dari logik aplikasi yang kita kembangkan. Sebagai contoh misalnya kita ingin selalu mengubah data dari sebuah kolom terlepas dari apakah data tersebut telah diganti oleh pihak lain atau tidak sejak terakhir kita

memperolehnya. Untuk melakukan hal tersebut kita dapat mengatur properti UpdateCheck pada atribut Column di kolom yang sesuai pada kelas entitas anda. Untuk HOL diatas anda dapat mengubah kode ini :

```
[Column(Storage="_Name", DbType="NVarChar(50)")]
    public string Name
{
```

menjadi :

```
[Column(Storage="_Name",
DbType="NVarChar(50)", UpdateCheck=UpdateCheck.Never)]
    public string Name
{
```

Dan kembali lakukan HOL dari langkah ke 3 maka anda tidak akan menemukan pesan kesalahan yang ada dari exception. Hal ini karena LINQ to SQL tidak akan mengevaluasi kolom Nama ketika melakukan pemeriksaan terhadap konkurensi. Nilai yang baru akan langsung dimasukkan ke dalam basis data dengan menimpa nilai yang sebelumnya.

Namun jika anda menginginkan perilaku lainnya maka anda dapat menggunakan cara lain. Ketika kita memanggil SubmitChanges, kita dapat menambahkan sebuah parameter yang berupa sebuah enum bernama ConflictMode untuk menentukan bagaimana perilaku dari Optimistic Concurrency yang digunakan oleh LINQ. Untuk enum ini ada dua pilihan yaitu :

1. ContinueOnConflict

Jika kita memilih opsi ini maka semua proses update ke basis data akan dilakukan, dan setiap kali menemui konflik yang menyangkut masalah konkurensi akan disimpan dalam properti ChangeConflicts pada DataContext terkait.

2. FailOnFirstConflict

Jika kita memilih opsi ini maka ketika pertama kali menemui konflik konkurensi ,proses update ke database akan langsung dihentikan.

Kemudian ketika kita memperoleh sebuah exception akibat dari timbulnya konflik konkurensi maka kita dapat menggunakan prosedur ResolveAll yang ada pada properti ChangeConflicts pada objek DataContext. Prosedur ini memiliki sebuah paramater yang berupa sebuah enum bernama RefreshMode dengan tiga anggota yang menentukan bagaimana perilaku LINQ dalam menyelesaikan konflik konkurensi saat melakukan update terhadap basis data:

1. KeepChanges

Jika kita memilih opsi ini maka nilai awal dari objek yang diperoleh dari database akan diganti dengan nilai yang baru. Namun semua perubahan yang telah kita buat akan tetap digunakan ketika kita melakukan update ke basis data.

2. KeepCurrentValues

Jika kita memilih opsi ini maka nilai di basis data akan diisi kembali dengan nilai yang sama seperti saat database dibaca pertama kali oleh aplikasi ini. kemudian perubahan yang telah kita buat pun akan tetap digunakan saat kita melakukan update ke basis data.

3. OverwriteCurrentValues

Jika kita memilih opsi ini maka semua nilai objek akan diisi dengan nilai yang baru di basis data. Semua perubahan yang kita buat akan ditulis ulang dengan data dari basis data.

Mari kita lihat contoh kasus dibawah:

	Nama	Alamat
Nilai Awal di basis data	Ronald	Bandung
Perubahan di luar aplikasi	Joko	Jakarta
Perubahan aplikasi	Mamat	

Dalam contoh kasus diatas tentunya aplikasi akan menemui masalah konkurensi karena setelah membaca nilai dari basis data, basis data mengalami perubahan. Maka ketika dalam aplikasi kita dilakukan pemanggilan ResolveAll perilaku dari masing-masing opsi adalah :

	Nama	Alamat
KeepChanges	Mamat	Jakarta
KeepCurrentValues	Mamat	Bandung
OverwriteCurrentValues	Ronald	Bandung

Seperti telah disebutkan sebelumnya diatas kita telah menggunakan Optimistic Concurrency, namun ada satu lagi tipe konkurensi yang juga bisa digunakan pada penanganan konkurensi yaitu Pessimistic Concurrency, hal ini dapat dilakukan dengan menggunakan konsep transaksi. Dengan menggunakan transaksi maka kita menerapkan proses locking pada basis data, sehingga ketika sebuah aplikasi sedang melakukan perubahan terhadap basis data aplikasi lainnya tidak dapat melakukan perubahan.

Hal diatas dapat dilakukan dengan menggunakan kelas TransactionScope yang ada dalam file DLL System.Transaction sehingga untuk menuliskan kode dibawah ini anda perlu mengekspor terlebih dahulu assembly System.Transaction. Untuk menggunakan model pessimistic concurrency maka kita perlu mengganti kode main kita menjadi seperti dibawah ini:

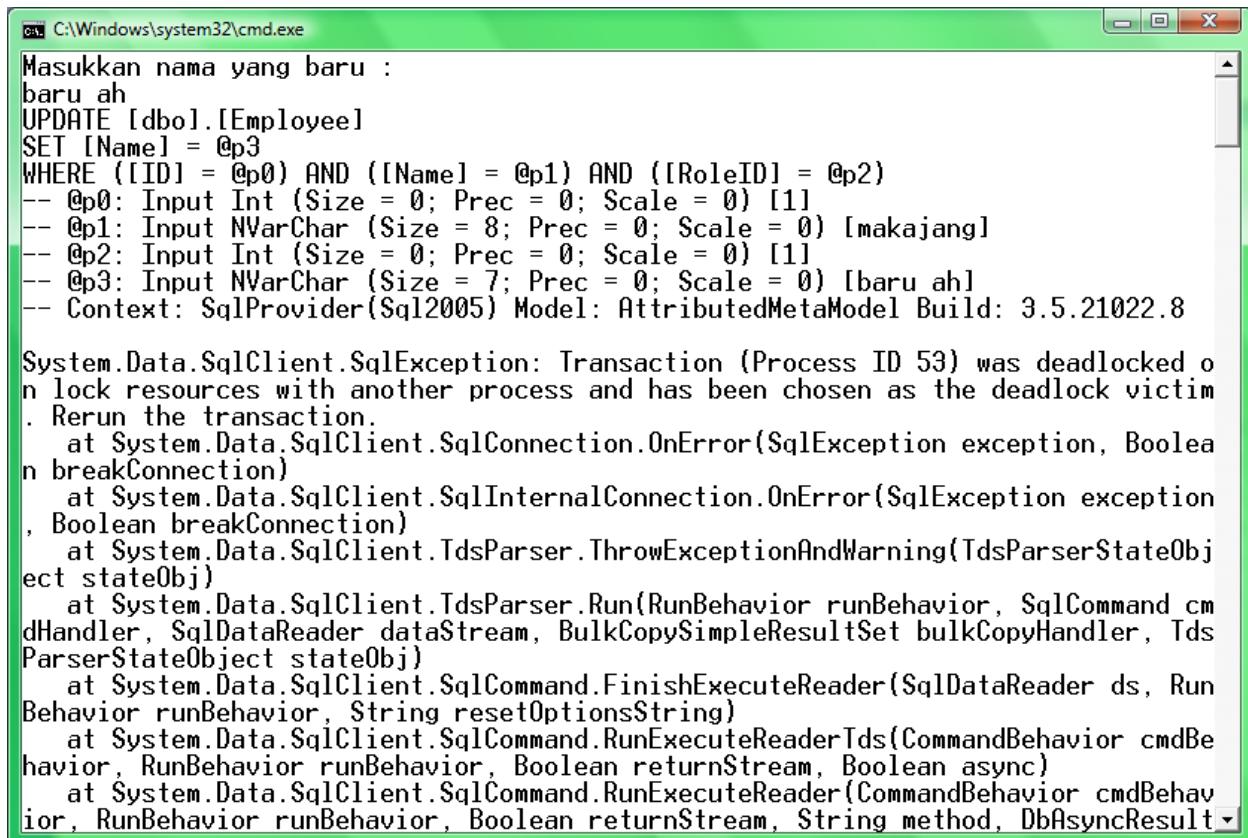
```
LINQDataClassesDataContext data = new LINQDataClassesDataContext();
using (TransactionScope t = new TransactionScope())
{
    Employee diganti = (from pegawai in data.Employees
                          where pegawai.ID == 1
                          select pegawai).First();
    Console.WriteLine("Masukkan nama yang baru :");
    string nama = Console.ReadLine();
    diganti.Name = nama;
    data.Log = Console.Out;
    try
    {
        data.SubmitChanges();
        t.Complete();
    }
    catch(SqlException SE)
```

```

    {
        Console.WriteLine(SE);
    }
}

```

Dalam kode diatas maka kita akan melakukan locking pada basis data. Sehingga jika misalnya kita mengulangi HOL dengan kode ini maka pada langkah ke 7 SQL Management Studio Express akan hang dan kemudian setelah kita memasukkan string pada langkah ke 8, SQL Management Studio Express akan kembali merespon dan diaplikasi kita akan ditampilkan pesan kesalahan. Seperti dibawah ini :



The screenshot shows a Windows Command Prompt window titled 'cmd C:\Windows\system32\cmd.exe'. The command entered was:

```

Masukkan nama yang baru :
baru ah
UPDATE [dbo].[Employee]
SET [Name] = @p3
WHERE ([ID] = @p0) AND ([Name] = @p1) AND ([RoleID] = @p2)
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [1]
-- @p1: Input NVarChar (Size = 8; Prec = 0; Scale = 0) [makajang]
-- @p2: Input Int (Size = 0; Prec = 0; Scale = 0) [1]
-- @p3: Input NVarChar (Size = 7; Prec = 0; Scale = 0) [baru ah]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

```

Following this, an exception stack trace is displayed:

```

System.Data.SqlClient.SqlException: Transaction (Process ID 53) was deadlocked o
n lock resources with another process and has been chosen as the deadlock victim
. Rerun the transaction.
   at System.Data.SqlClient.SqlConnection.OnError(SqlException exception, Boolean
n breakConnection)
   at System.Data.SqlClient.SqlInternalConnection.OnError(SqlException exception
, Boolean breakConnection)
   at System.Data.SqlClient.TdsParser.ThrowExceptionAndWarning(TdsParserStateObj
ect stateObj)
   at System.Data.SqlClient.TdsParser.Run(RunBehavior runBehavior, SqlCommand cmd
Handler, SqlDataReader dataStream, BulkCopySimpleResultSet bulkCopyHandler, Tds
ParserStateObject stateObj)
   at System.Data.SqlClient.SqlCommand.FinishExecuteReader(SqlDataReader ds, Run
Behavior runBehavior, String resetOptionsString)
   at System.Data.SqlClient.SqlCommand.RunExecuteReaderTds(CommandBehavior cmdBe
havior, RunBehavior runBehavior, Boolean returnStream, Boolean async)
   at System.Data.SqlClient.SqlCommand.RunExecuteReader(CommandBehavior cmdBehav
ior, RunBehavior runBehavior, Boolean returnStream, String method, DbAsyncResult)

```

Hal ini terjadi karena telah terjadi deadlock dimana aplikasi kita dan juga SQL Management Studio Express sama-sama mengunci basis data, sehingga pada akhirnya SQL Management Studio Express yang memperoleh hak untuk mengubah basis data dan aplikasi kita akan memperoleh pesan error.

Penutup

Pada akhir perjalanan kita di desa LINQ to SQL kita sekali lagi telah diperlihatkan bagaimana kemampuan LINQ untuk secara baik beroperasi terhadap basis data dengan menggunakan LINQ to SQL. Kemudian telah ditanganinya isu konkurensi dengan LINQ serta integrasinya yang baik dengan ADO .NET saat kita melakukan transaksi pun telah menjadi poin penting yang membuat anda perlu mulai memikirkan untuk penggunaan LINQ ketika berhubungan dengan basis data. Selain tentunya kemudahan penulisan query dan tool-tool menarik yang memudahkan penggeraan.

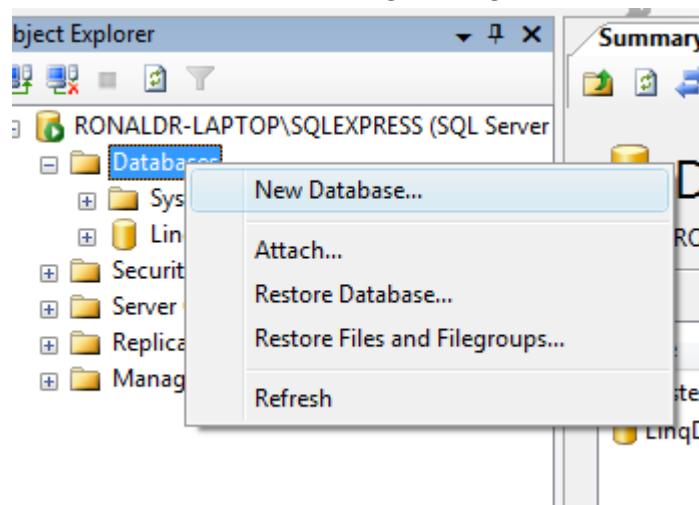
LAMPIRAN A

Pembuatan basis data untuk HOL operasi LINQ to SQL

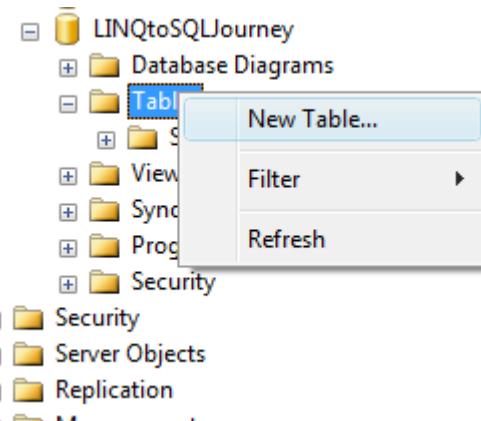
Task 1 : Membuat basis data

Dalam task ini anda akan membuat sebuah basis data dengan menggunakan SQL Server 2005.

1. Buka SQL Server Management Studio Express 2005.
2. Buat sebuah database baru dengan mengklik Tab **Database** kemudian pilih **New Database ...**



3. Beri nama "LINQtoSQLJourney" kemudian Klik OK.
4. Pada bagian LINQtoSQLJourney klik di Tables kemudian klik kanan dan pilih **New Table ...**



5. Buat sebuah tabel dengan skema seperti berikut :

Table - dbo.Table_1* Summary

Column Name	Data Type	Allow Nulls
ID	int	<input type="checkbox"/>
Name	nvarchar(50)	<input checked="" type="checkbox"/>
Address	text	<input checked="" type="checkbox"/>
RoleID	int	<input checked="" type="checkbox"/>
		<input type="checkbox"/>

Catatan :

1. Kolom ID adalah primary key, sehingga anda perlu mengklik kanan disamping nama kolom ID dan memilih opsi Set Primary Key.
2. Kolom ID pun memiliki sifat auto increment sehingga untuk properti Identity Specification harus diisi dengan nilai "Yes".

Column Properties

Full-text Specification	No
Has Non-SQL Server Subscriber	No
Identity Specification	Yes
(Is Identity)	Yes
Identity Increment	1
Identity Seed	1
Todavable	Var
(Is Identity)	

6. Simpan tabel tersebut dan beri nama "Employee"
7. Buat sebuah tabel baru lagi dengan skema seperti berikut :

Table - dbo.Role Summary

Column Name	Data Type	Allow Nulls
ID	int	<input type="checkbox"/>
RoleName	nvarchar(50)	<input checked="" type="checkbox"/>
RoleDescription	text	<input checked="" type="checkbox"/>

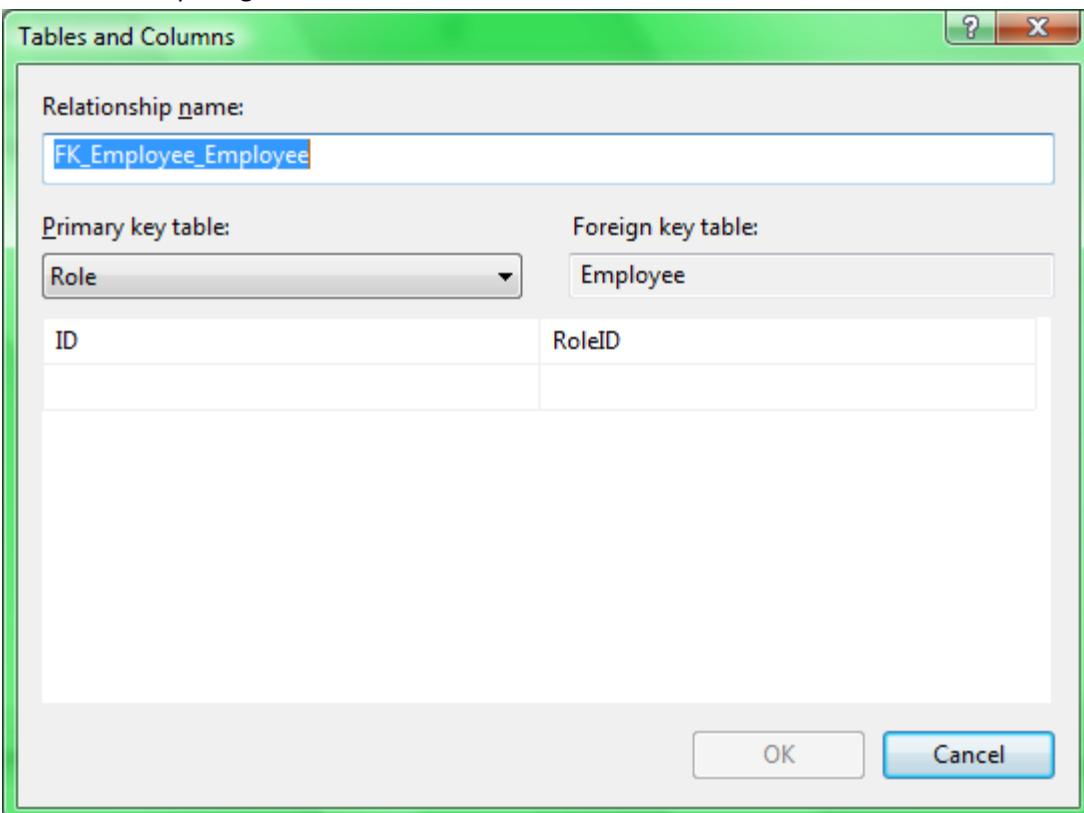
Catatan :

1. Kolom ID adalah primary key dan juga memiliki sifat auto increment. Cara pembuatannya sama seperti langkah 5.
8. Simpan tabel tersebut dan beri nama "Role"
9. Buat sebuah relasi foreign key pada tabel Employee, caranya adalah dengan melakukan klik kanan pada jendela desain tabel Employee. Kemudian pilih Relationship...

Table - dbo.Employee Summary

Column Name	Data Type	Allow
ID	int	
Name		<input checked="" type="checkbox"/>
Address		<input checked="" type="checkbox"/>
RoleID		<input checked="" type="checkbox"/>
<input type="button" value="Relationships..."/>		
<input type="button" value="Indexes/Keys..."/>		
<input type="button" value="Fulltext Index..."/>		
<input type="button" value="XML Indexes..."/>		
<input type="button" value="Check Constraints..."/>		
<input type="button" value="Generate Change Script..."/>		

- Pada bagian Tables And Columns Spesification tekan tombol elipsis (...) sehingga nampak jendela seperti dibawah ini :
- Pilih Role sebagai primary key table dan field ID, kemudian untuk foreign table kita masukkan field RoleID. Seperti gambar dibawah ini :



- Tekan OK dan basis data anda telah siap. Namun sebagai tambahan silahkan isi data dibawah ini pada tabel Role.

	ID	RoleName	RoleDescription
	1	Director	Direktur Perusahaan
	2	Programmer	Staff programmer
	3	Desainer	Staff desain
	4	Office Boy	Pesuruh di kantor
▶*	NULL	NULL	NULL

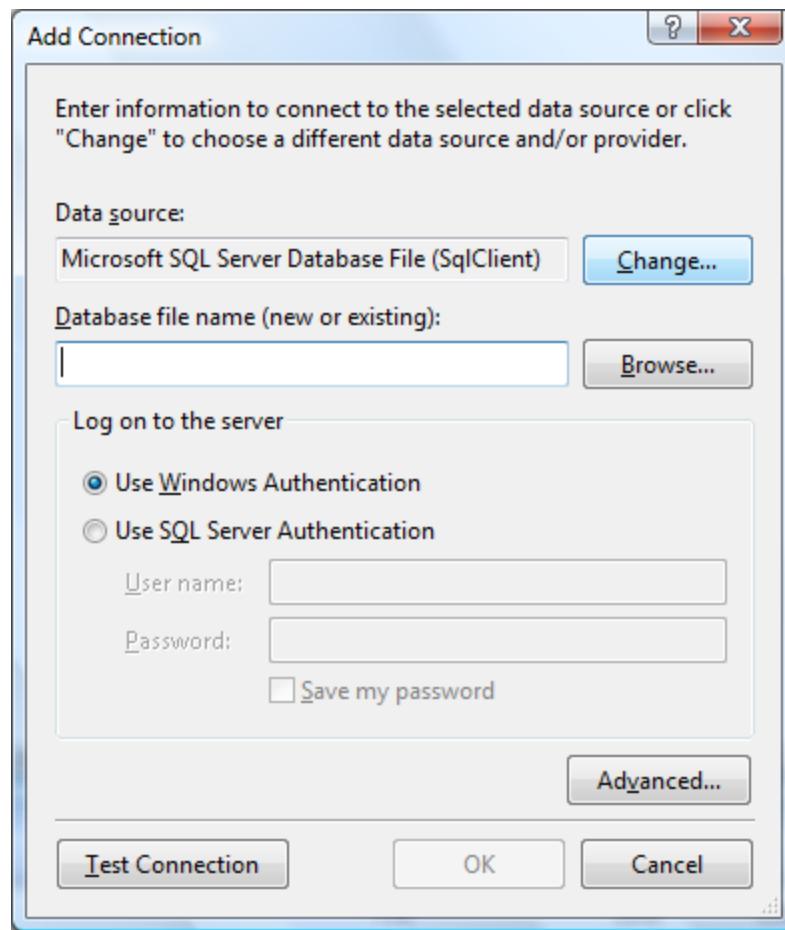
13. Kemudian isikan data dibawah ini pada tabel Employee

	ID	Name	Address	RoleID
	1	Ronald	Bandung	1
	2	Joko	Ciamis	4
	3	Fajar	Bandung	2
	4	Haris	Jakarta	3
	5	Mamat	Bandung	2
▶*	NULL	NULL	NULL	NULL

Task 2 : pembuatan project

Dalam task ini anda akan membuat sebuah project yang berisikan kelas entitas yang mewakili database pada task 1.

1. Mulai Visual Studio C# Express 2008 anda. (jika anda menggunakan vista sebaiknya jalankan VS sebagai administrator dengan memilih opsi run as administrator).
2. Buat sebuah project baru dengan tipe console application beri nama “LinqToSQLJourney”
3. Buka jendela “Database Explorer” anda dapat melakukannya secara cepat dengan menekan kombinasi tombol Ctrl+W, L.
4. Klik kanan pada bagian “Data Connection” kemudian pilih “Add Connection ...” pada menu popup yang muncul.
5. Pada bagian Datasource pilih “Microsoft SQL Server Database File” setelah itu anda akan dihadapkan pada jendela seperti gambar dibawah.

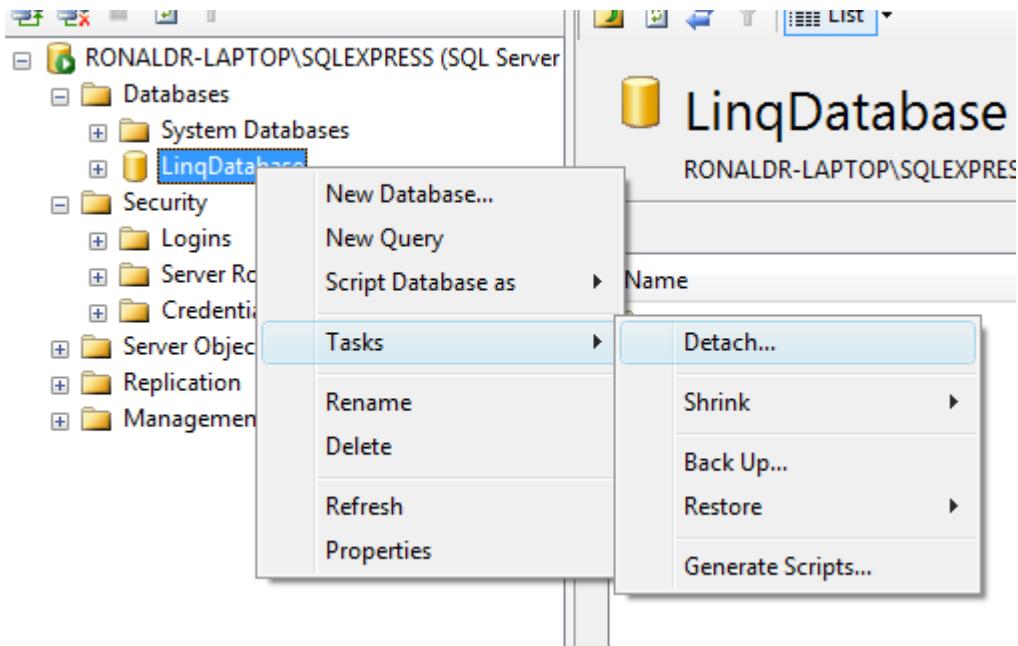


Gambar 1 jendela Add Connection

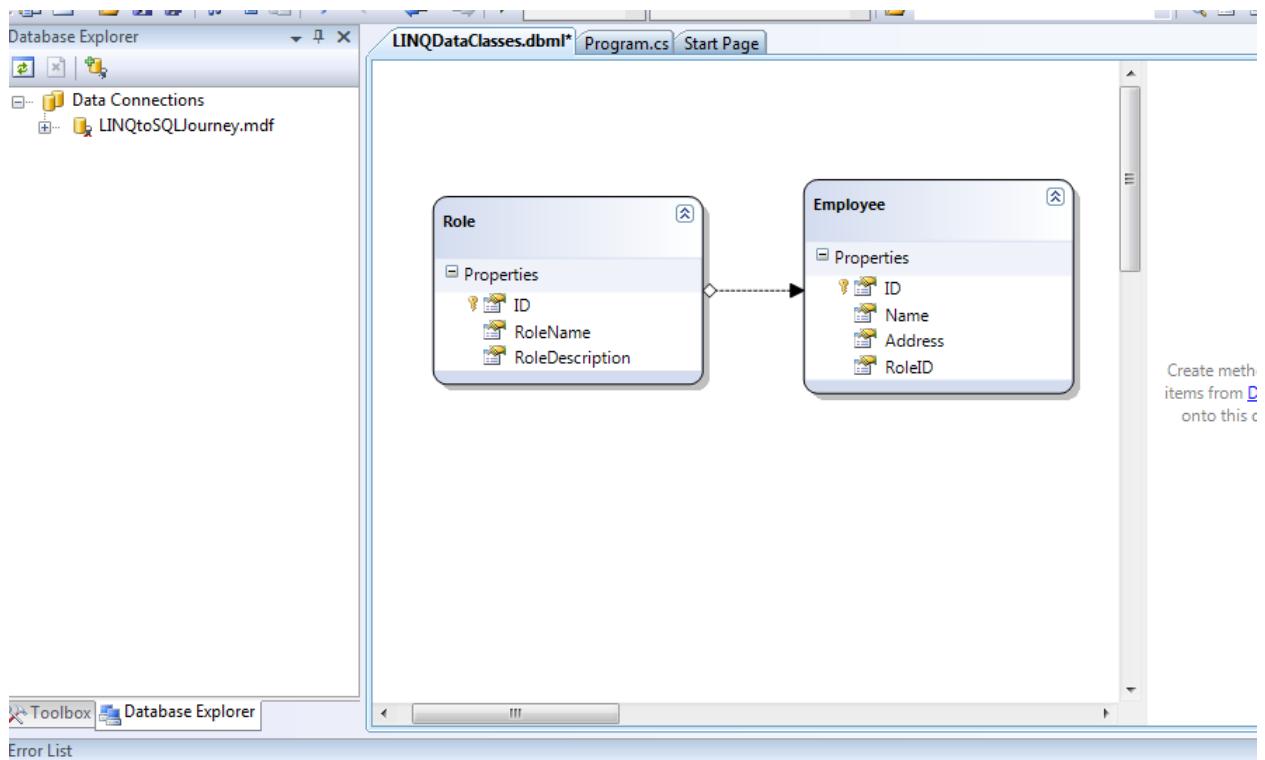
6. Pada bagian “Database file name (new or existing)” arahkan pada file mdf database yang diinginkan (pada komputer penulis lokasi file tersebut ada di C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Data\ LINQTutorial.mdf).

Catatan :

Jika muncul pesan error yang menandakan bahwa file tersebut sedang dipakai maka pada SQL Management Studio 2005 anda perlu mendeattach file tersebut seperti pada gambar dibawah ini:



7. Pilih OK jika ada jendela yang menanyakan apakah anda akan mengkopi file dengan ekstensi "MDF".
8. Tambahkan file baru pada projek yang kita buat, caranya cukup mudah anda hanya perlu menekan kombinasi tombol Ctrl+Shift+A. Kemudian anda akan dihadapkan pada jendela seperti pada gambar 4.
9. Pilih "LINQ to SQL Classes" dan beri nama "LINQDataClasses.dbml"
10. Kemudian anda akan dihadapkan pada jendela "Object Relational Designer", jendela ini merupakan jendela yang akan membantu anda dalam membuat sebuah objek yang akan melakukan mapping antara data yang ada di basis data relational kepada objek.
11. Tarik tabel Employee dan tabel Role pada jendela "Database Explorer" ke jendela "Object Relational Designer" kemudian tampilan di jendela anda akan seperti gambar dibawah ini. Dibalik layar sebenarnya Visual Studio secara otomatis akan membangun sebuah kelas turunan dari DataContext yang memetakan data relasional kita ke objek. Kode dari kelas ini dapat dilihat pada file "LINQDataClasses.designer.cs" di solusi kita.



12. Sekarang anda telah selesai menyiapkan sebuah project yang berisikan kelas datacontext yang mewakili tabel di basis data.

Task 3 : penyesuaian project agar dapat membaca data dari server

Jika anda jeli pada task 2 diatas anda akan memperhatikan bahwa Visual Studio Express akan mengkopi file database yang berekstensikan .mdf ke dalam project anda. Sehingga sebenarnya operasi terhadap database bukan ditujukan terhadap data yang ada diserver, melainkan terhadap data yang ada di file yang dikopi tersebut. Hal ini karena Database Explorer dari Visual Studio Express tidak mendukung data source yang bertipe "Microsoft SQL Server", sehingga kita perlu melakukan sedikit *tweak* agar operasi database kita ditujukan ke basis data yang ada di sql server express.

1. Hapus file LINQtoSQLJourney yang ada dalam project anda.
2. Buka file "app.config".
3. Ganti atribut connectionString pada elemen add di dalam file XML sehingga menjadi seperti berikut :

```

<add
  name="LinqToSQLJourney.Properties.Settings.LINQtoSQLJourneyConnectionString"
  connectionString="Data Source=localhost\SQLEXPRESS;Initial Catalog=LINQtoSQLJourney;Integrated Security=True;" providerName="System.Data.SqlClient" />

```

langkah diatas akan mengganti connectionstring yang asalnya mengarahkan basisdata ke file mdf yang ada dalam project anda menjadi diarahkan ke SQL Server express yang terinstall di komputer anda.

LAMPIRAN B

Kode inisialisasi kelas Pegawai dan Role

```
public class Pegawai
{
    public int IdPelanggan { get; set; }
    public string Nama { get; set; }
    public string Alamat { get; set; }
    public int IDJabatan { get; set; }
    public Pegawai(int ID)
    {
        IdPelanggan = ID;
    }

    public override string ToString()
    {
        return Nama + "\t" + Alamat + "\t" + IdPelanggan;
    }
    public static List<Pegawai> createPegawai()
    {
        return new List<Pegawai>
        {
            new Pegawai(1) { Nama = "Ronald", Alamat= "Bandung", IDJabatan=1 },
            new Pegawai(2) { Nama = "Fuady", Alamat= "Condet", IDJabatan=1 },
            new Pegawai(3) { Nama = "Zeddy", Alamat= "Jakarta", IDJabatan=2 },
            new Pegawai(4) { Nama = "Narendra", Alamat= "Jakarta", IDJabatan=2 },
            new Pegawai(5) { Nama = "Umar", Alamat= "Bandung", IDJabatan=2 },
            new Pegawai(6) { Nama = "Arief", Alamat= "Bandung", IDJabatan=3 },
            new Pegawai(7) { Nama = "Fajar", Alamat= "Bandung", IDJabatan=4 },
            new Pegawai(8) { Nama = "Anggriawan", Alamat= "Bandung", IDJabatan=3 }
        };
    }
}

class Jabatan
{
    public int ID { get; private set; }
    public string NamaJabatan{ get; set; }
    public string DeskripsiJabatan { get; set; }
    public Jabatan(int ID)
    {
        this.ID = ID;
    }
    public override string ToString()
    {
```

```
        return "ID = "+ID+"\t NamaJabatan = "+NamaJabatan+"\t Deskripsi  
:"+DeskripsiJabatan;  
    }  
    public static List<Jabatan> createJabatan()  
{  
    return new List<Jabatan>()  
    {  
        new Jabatan(1) {NamaJabatan="MIC  
Lead", DeskripsiJabatan="Memimpin lab MIC"},  
        new Jabatan(2) {NamaJabatan="DPE  
Team", DeskripsiJabatan="Pegawai Microsoft"},  
        new Jabatan(3) {NamaJabatan="MIC Crew", DeskripsiJabatan="Kru  
tetap MIC"},  
        new Jabatan(4) {NamaJabatan="MIC Infrastructure  
Architect", DeskripsiJabatan="Bertanggung jawab akan infrastruktur IT di MIC"}  
    };  
}
```