# The LINQ Project

## .NET Language Integrated Query

**September 2005 (Original)**

**April 2006 (Transated)**

Don Box, Architect, Microsoft Corporation and
Anders Hejlsberg, Distinguished Engineer, Microsoft Corporation

Русский перевод и дополнения: Скрипкин П. В., Microsoft Corporation (Россия)

Издание специально отредактировано для Конференции VBStreets (http://bbs.vbstreets.ru)

## Что такое LINQ, и для чего он нужен

Аббревиатура LINQ расшифровывается как “Language INtegrated Query”. Это сочетание буквально переводится «Запросы, интегрированные в язык». Что же эта технология из себя представляет, и для чего она нужна? В этой статье я постараюсь ответить на этот и на многие другие вопросы. По пути повествования я также расскажу, что ещё, кроме LINQ, нас ждёт в Visual Basic 9.0 и C# 3.0.

На данном этапе развития объектно-ориентированных технологий наиболее сложным и проблематичным является доступ к информации, которая изначально не определена объектами. Двумя наиболее распространёнными источниками подобной информации являются базы данных и XML. Оговорюсь: существуют объектные базы данных, но их рассмотрение и интеграция с .NET выходит за рамки данного обзора.

Что же позволяет нам LINQ? LINQ, как это видно из названия, является технологией, которая позволяет использовать язык запросов, похожий с SQL, прямо в коде программы. LINQ будет поддерживаться компиляторами Visual Basic 9.0 и C# 3.0 (а точнее, уже поддерживается VB .NET 2005 и C# 2.0 при установке дополнительных компонент, которые мы обсудим отдельно); относительно его, так сказать, *встроенной* поддержки другими языками .NET (J# и C++/CLI) пока что достоверных сведений нет. Впрочем, в любом случае, поддерживаться он всё равно будет, как ни крути, но гораздо более кривым способом. Давайте же рассмотрим LINQ более детально.

LINQ определяет набор стандартных операторов запросов (*standard query operators*), которые могут использоваться для фильтрации, выборки и сортировки элементов, определённых типом `IEnumerable<T>` (перечислений).

---

**Примечание**

Под *перечислением* здесь и далее понимается объект, реализующий IEnumerable<T>, а не перечисление в привычном понимании смысла этого слова (enumeration).

---

Таким образом, мы можем фильтровать содержимое любых коллекций, массивов, да и вообще любых перечислимых объектов с помощью такого синтаксиса:

Код C# 3.0

```
string[] users = {"RayShade", "tyomitch", "GSerg"};
IEnumerable<string> coolusers = from user in users
                                where user.Length == 7
                                select user.ToUpper();
```

Код VB .NET 9.0

```
Dim users() As String = {"RayShade", "tyomitch", "GSerg"}
Dim coolusers As IEnumerable(Of String) = Select user.ToUpper() _
                                          From user In users _
                                          Where user.Length = 7
```

Как можно заметить, у C# и VB несколько отличается порядок ключевых слов: в Visual Basic 9.0 используется более традиционный вариант (**`Select … From … Where … Order By …`**), позаимствованный из SQL, в то время, как в C# используется новый вариант (**`from … where … orderby … select …`**). Дискуссию по поводу этого вопроса можно почитать здесь. Впрочем, это только дело вкуса.

Итак, давайте вернёмся к LINQ. Как я уже сказал, LINQ определяет стандартные операторы запросов, такие, как Select, Where, и т.п. Однако, на этом дело не заканчивается. Вы также можете определять собственные операторы запросов (*query operators*) и переопределять существующие. Какой смысл в переопределении? Например, Dlinq (технология LINQ для доступа к БД) транслирует команды запросов языка в аналогичные запросы T-SQL. Более подробно о переопределении написано ниже. Впрочем, перед тем, как кликать по ссылке, советую всё-таки почитать начало :))

Благодаря расширяемости LINQ появились две его реализации для эффективного и удобного взаимодействия с реляционными данными (DLinq) и с XML (XLinq). Их рассмотрение мы оставим на чуть более позднее время, скажу только, что мне они кажутся куда более полезными, чем сам по себе LINQ. Короче говоря, дорогие читатели, придержите своё любопытство и прочитайте эту часть статьи, где рассказывается о LINQ в общем и целом.





3

## Стандартные операторы запросов

Поскольку всё проще понять на примере, вот довольно простой пример использования LINQ. Попробуйте сами угадать, что он выводит:

Код C# 3.0

```
using System;
using System.Query;
using System.Collections.Generic;

class Program {
     static void Main() {
           string[] users = {"RayShade", "tyomitch", "GSerg"};
           IEnumerable<string> coolusers = from user in users
                                           where user.Length == 8
                                           orderby user
                                           select user.ToUpper();
           foreach (string user in coolusers)
                 Console.WriteLine(user);
           Console.ReadLine();
     }
}
```

Код VB .NET 9.0

```
Imports System
Imports System.Query
Imports System.Collections.Generic

Module Program

    Sub Main()
        Dim users() As String = {"RayShade", "tyomitch", "GSerg"}
        Dim coolusers As IEnumerable(Of String) = _
                                  Select user.ToUpper() _
                                  From user In users _
                                  Where user.Length = 7 _
                                  Order By user
        For Each user As String In coolusers
            Console.WriteLine(user)
        Next
        Console.ReadLine()
    End Sub

End Module
```

Соответственно, результатом выполнения данной программы будет следующее:

```
RAYSHADE
```

Чтобы понять, как работает LINQ, давайте ещё раз рассмотрим оператор запроса нашей программы:

Код C# 3.0

```
IEnumerable<string> coolusers = from user in users
                                where user.Length == 7
                                orderby user
                                select user.ToUpper();
```

Код VB .NET 9.0

```
Dim coolusers As IEnumerable(Of String) = _
                                   Select user.ToUpper() _
                                   From user In users _
                                   Where user.Length = 7 _
                                   Order By user
```

Здесь переменная coolusers инициализируется выражением запроса (*query expression*). Скажу по секрету, на самом деле она инициализируется типом `System.Query.OrderedSequence<string, string>`, но пока что нам это довольно мало говорит :) В данном примере используются стандартные операторы запросов, в частности **`where, orderby`** и **`select`**.

И C#, и Visual Basic поддерживают так называемый синтаксис запросов (*query syntax*). Подобно операторам `using` и `foreach`, которые на самом деле являются сокращённым вариантом довольно громоздкой записи, синтаксис запросов также является лишь упрощённым вариантом вот такой записи:

Код C# 3.0

```
IEnumerable<string> coolusers = users
                            .Where(f => f.Length == 7)
                            .OrderBy(f => f)
                            .Select(f => f.ToUpper());
```

**Примечание**

Некоторых почему-то пугает такой тип записи, когда каждая точка находится на разной строке. Поясню: этот код можно записать и как

Код C# 3.0

```
users.Where(f => f.Length == 7).OrderBy(f => f).Select(f => f.ToUpper());
```

Просто такой вариант менее читаем.

Аргументы, переданные методам (мы их будем называть *операторами запросов*, или просто *операторами*) `Where`, `OrderBy` и `Select`, называются λ-выражениями (*lambda expressions*, произносится «лямбда-выражения»), которые являются некоторого рода эволюцией анонимных делегатов, позволяя их записать в ещё более сокращённой форме. О них мы ещё поговорим чуть ниже. Именно компактность λ-выражений позволяет использовать их в тексте запроса без существенного увеличения объёма кода.





5

# Функции языка, необходимые для LINQ

Language Integrated Query построен на базе функций языка, многие из которых являются новыми для VB .NET и C#. Давайте же рассмотрим их.

## *Деревья выражений и λ-выражения*

**Примечание**

Поскольку λ-выражения для VB .NET ещё не реализованы в текущем Community Technology Preview, примеры кода для него не даны. Статья будет обновлена с примерами кода, как только это станет возможным.

Как было сказано раннее, λ-выражения являются ещё более мощным способом сократить объём кода, чем анонимные методы. Например, λ-выражение

Код C# 3.0

```
user => user.Length == 7
```

эквивалентно анонимному методу

Код C# 3.0

```
delegate (string user) {return user.Length == 7;}
```

или обычному

Код C# 3.0

```
bool MyLambda (string user)
{
     return user.Length == 7;
}
```

Код VB .NET 9.0

```
Function MyLambda (user As String)
     Return user.Length = 7
End Function
```

Но это ещё не всё. λ-выражения приводятся к делегату `System.Query.Func<>`, который может быть представлен в следующем виде:

- 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, A2, A3, T>**(**A0** *arg0*, **A1** *arg1*, **A2** *arg2*, **A3** *arg3*)

Таким образом, код, который мы использовали для запроса, можно переписать так:

Код C# 3.0

```
Func<string, bool>   filter  = (user => user.Length == 7);
Func<string, string> extract = (user => user);
Func<string, string> project = (user => user.ToUpper());

IEnumerable<string> expr = names.Where(filter)
                                .OrderBy(extract)
                                .Select(project);
```

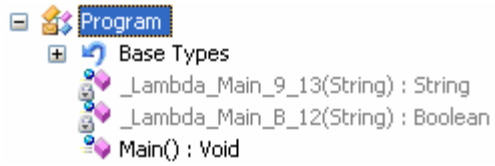
Без использования λ-выражений этот код выглядел бы даже так:

Код C# 3.0

```
Func<string, bool>   filter  = delegate (string user) {
                                   return user.Length == 7;
                               };

Func<string, string> extraction = delegate (string user) {
                                   return user;
                               };

Func<string, string> projection = delegate (string user) {
                                   return user.ToUpper();
                               };

IEnumerable<string> expr = names.Where(filter)
                                .OrderBy(extraction)
                                .Select(projection);
```

На самом деле компилятору абсолютно всё равно, какой синтаксис вы используете. Более того, неужели вы думаете, что я смог удержаться и не глянуть Reflector'ом на то, что генерирует компилятор, когда встречает λ-выражения.

Смотрите сами.




В нашем классе появились два метода, названные соответственно `_Lambda_Main_9_13` и `_Lambda_Main_B_12`. Как нетрудно догадаться, это и есть наши λ-выражения, которые компилятор превратил в private-методы. Однако, это не единственный способ использовать λ-выражения.

С приходом LINQ появилось новое пространство имён `System.Expressions`. В нём определён тип `Expression<T>`, который позволяет строить деревья выражений (*expression trees*). Вся суть в том, что при присвоении λ-выражения переменной типа `Expression<T>` метод не генерируется: λ-выражение сохраняется в бинарном виде. Вот что сказано по этому поводу в MSDN: *Expression trees are efficient in-memory data representations of lambda expressions and make the structure of the expression transparent and explicit*. Возникает здравый вопрос: зачем же это нужно? Всё очень просто. При проектировании DLinq было необходимо преобразовывать λ-выражение, переданное оператору `Where`, в SQL-запрос. Разумеется, тело метода трудно обратно декомпилировать и транслировать в SQL, поэтому и ввели деревья выражений.

Допустим, у нас есть два выражения:

Код C# 3.0

```
Func<string, bool>            isVBStreetsFunc = (s => s == "VBStreets");
Expression<Func<string,bool>> isVBStreetsExpr = (s => s == "VBStreets");
```

В таком случае, этот код скомпилируется:

Код C# 3.0

```
bool isVBStreets = isVBStreetsFunc("NotVBStreets");
```

А этот вызовет ошибку, т.к. выражения являются всего лишь данными и не могут выполняться:

Код C# 3.0

```
bool isVBStreets = isVBStreetsExpr("NotVBStreets");
```

Лично меня очень впечатлила возможность вот такого «разбора» λ-выражения:

Код C# 3.0

```
Expression<Func<string,bool>> isVBStreets = (s => s == "VBStreets");
Console.WriteLine("Number of parameters: {0}", isVBStreets.Parameters.Count);
foreach (ParameterExpression p in isVBStreets.Parameters)
    Console.WriteLine("    Parameter {0} of type {1};", p.Name, p.Type);
Console.WriteLine("Return type: {0}", isVBStreets.Body.Type);
MethodCallExpression body = (MethodCallExpression)isVBStreets.Body;
Console.WriteLine("Expression calls {0} method", body.Method.Name);
ConstantExpression secondPart = (ConstantExpression)body.Parameters[1];
Console.WriteLine("Second part of expression: {0}", secondPart.Value);
Console.WriteLine("Expression.ToString method: {0}", isVBStreets);
```

Этот код выводит

```
Number of parameters: 1
    Parameter s of type System.String;
Return type: System.Boolean
Expression calls op_Equality method
Second part of expression: VBStreets
Expression.ToString method: |s| op_Equality(s, "VBStreets")
```

## *Методы-расширения*

Для начала нам необходимо познакомиться с методами-расширениями (*extension methods*). Метод-расширение позволяет нам добавить свой метод ко всем классам. Выглядит примерно так:

Код C# 3.0

```
namespace MyExtensions
{
    using System.Runtime.CompilerServices;

    public static class MyExtension
    {
        [Extension]
        public static string GetTypeFullName(this object source)
        {
            return source.GetType().FullName;
        }

        [Extension]
        public static int MakeDouble(this int num)
        {
            return num * 2;
        }

    }
}
```

Использование:

Код C# 3.0

```
namespace ExtensionsTest
{
    using MyExtensions;
    class Program
    {
        static void Main(string[] args)
        {
            object o = new object();
            int i = 5;
            Console.WriteLine(o.GetTypeFullName());
            //Console.WriteLine(MyExtension.GetTypeFullName(o));
            Console.WriteLine(i.MakeDouble());
            //Console.WriteLine(MyExtension.MakeDouble(i));
            Console.Read();
        }
    }
}
```

Короче, весьма удобная штука. Но у них ещё одна функция – именно они позволяют нам использовать операторы LINQ. По сути, стандартные операторы `Select`, `Where`, `OrderBy` являются методами-расширениями, определёнными в классе `System.Query.OrderedSequence` (где-то мы его видели?). Давайте взглянем на методы-расширения более детально. Вот несколько их особенностей:

- Классы, определяющие методы-расширения, должны быть статическими.
- Методы-расширения должны быть объявлены с атрибутом `[System.Runtime.CompilerServices.Extension]`.





9

- В метод-расширение всегда передаётся параметр, помеченный ключевым словом `this` – ссылка на экземпляр класса, к которому это расширение применяется. Кстати, тип объекта, к которому расширение можно применить, можно ограничить, просто определив тип этого параметра, отличный от `object`, как показано в примере с `MakeDouble()`.
- Для импорта методов-расширений, определённых в каком-то классе, необходимо объявить пространство имён, в котором находится этот класс, в директиве `using`.

После того, как мы ознакомились с концепцией методов-расширений в общих чертах, давайте взглянем на конкретный пример реализации запроса:

Код C# 3.0

```
namespace System.Query
{
  using System;
  using System.Collections.Generic;

  public static class Sequence {
    public static IEnumerable<T> Where<T>(
              this IEnumerable<T> source,        //Источник - перечисление
                   Func<T, bool> predicate) {    //Функция-условие

      foreach (T item in source)                 //Для каждого объекта в источнике
        if (predicate(item))                     //Если условие выполняется
          yield return item;                     //Добавить в перечисление-результат
    }
  }
}
```

Как видно, тип объекта, к которому применяется расширение, ограничен типом `IEnumerable<T>` (перечислением). Теперь мы можем написать такой код:

Код C# 3.0

```
IEnumerable<string> coolusers = users.Where(f => f.Length == 7);
```

Он будет эквивалентным следующему коду:

Код C# 3.0

```
IEnumerable<string> coolusers = Sequence.Where(users,
                                        f => f.Length == 7);
```

Поскольку, чтобы компилятор «заметил» присутствие статических типов с методами-расширениями, необходимо объявить их в директиве `using` (`Imports` в VB), для использования стандартных операторов запросов, как нетрудно догадаться, необходимо написать

Код C# 3.0

```
using System.Query; // делает видимыми операторы запросов
```

Код VB .NET 9.0

```
Imports System.Query // делает видимыми операторы запросов
```

Все операторы запросов работают с перечислениями `IEnumerable<T>`, кроме оператора `OfType`, который работает с перечилениями в стиле 1.0/1.1 фреймворка, «приводя их в порядок», то есть к виду `IEnumerable<T>`. Разумеется, он также может применяться и к перечислениям вида `IEnumerable<T>`, позволяя выбирать из перечислений только объекты определённого типа.

Вот пример приведения коллекции из вида 1.0 к виду `IEnumerable<T>`:

Код C# 3.0

```
// коллекция в «классическом» виде не может использоваться с запросами...
IEnumerable classic = new OlderCollectionType();

//... поэтому мы представим её в виде IEnumerable<object>
IEnumerable<object> modern = classic.OfType<object>();
```

Вот пример другого использования оператора `OfType`:

Код C# 3.0

```
IEnumerable<object> stuff = new object[]{"VBStreets", 1, true, "RSDN"};
IEnumerable<string> users = stuff.OfType<string>(); //отбираем только строки
```

Собственно говоря, вот стандартная реализация оператора `OfType`.

Код C# 3.0

```
public static IEnumerable<T> OfType<T>(this IEnumerable source) {
  foreach (object item in source)
    if (item is T)
      yield return (T)item;
}
```

## *Отложенная обработка запроса*

Как можно заметить, стандартный оператор `Where` реализован с использованием конструкции `yield return`, впервые появившейся в C# 2.0. На самом деле, все операторы запросов используют её. Таким образом, используется так называемая отложенная обработка запроса (*deferred query evaluation*): результат запроса вычисляется не при его объявлении, а только при самом перечислении. Это несложно проверить, написав небольшой бенчмарк, но мы не будем сейчас тратить на это время. Что самое интересное, один и тот же запрос может давать разные результаты в разные моменты времени. Давайте взглянем на один простенький примерчик, который это демонстрирует:





11

Код C# 3.0

```
class Program {
     static void Main() {
           List<string> users = new List<string>();
           users.AddRange(new string[]{«RayShade», «tyomitch», «GSerg»});
           IEnumerable<string> coolusers = from user in users
                                           where user.Length == 7
                                           select user; //должен быть VBStreets
           users.Add("Hmmm..."); //добавим ещё один элемент
           foreach (string user in coolusers)
                 Console.WriteLine(user);
           Console.ReadLine();
     }
}
```

Код VB .NET 9.0

```
Module Program

     Sub Main()
          Dim users As New List(Of String)
          users.AddRange(New String() {«RayShade», «tyomitch», «GSerg»})
          Dim coolusers As IEnumerable(Of String) = Select user _
              From user In users _
              Where user.Length = 7 'должен быть VBStreets
          users.Add("Hmmm...") 'добавим ещё один элемент
          For Each user As String In coolusers
             Console.WriteLine(user)
          Next
         Console.ReadKey()
     End Sub

End Module
```

И... о, чудо!

```
VBStreets
Hmmm...
```

Это показывает, что выражение запроса вычисляется не во время его объявления, а во время его выполнения. Однако, это порождает одну проблему. Допустим, мы пишем приложение, работающее с БД, и при загрузке формы мы выполняем запрос к БД, а его результат записываем в переменную-перечисление `IEnumerable<T>`. Затем, каждый раз, когда пользователь активизирует форму, необходимо очистить `DataGridView` и вновь заполнить его элементами из этого перечисления. Ещё не заметили потенциальную проблему? Каждый раз, когда мы перебираем элементы этого перечисления, приложение вновь обращается к базе данных и выполняет запрос! К счастью, если `IEnumerable<T>` преобразовать в `List<T>` или в массив, запрос будет выполняться один раз:

Код C# 3.0

```
IEnumerable<string> coolusers = (from user in users
                                 where user.Length == 7
                                 select user).ToArray(); //или .ToList();
```

Код VB .NET 9.0

```
Dim coolusers As IEnumerable(Of String) = (Select user _
    From user In users _
    Where user.Length = 7).ToArray()'или .ToList()
```

Вот так-то.

## *Упрощённая инициализация объектов*

Зачастую бывает нужно инициализировать тип с некоторыми заранее заданными значениями свойств, отличными от значений по умолчанию. Допустим, у нас есть класс `User`:

Код C# 3.0

```
public class User {
  string _name;
  string _admin;
  string _url;
  int _membersCount;

  public string Name {
    get { return _name; } set { _name = value; }
  }

  public string Admin {
    get { return _admin; } set { _admin = value; }
  }

  public string URL {
    get { return _url; } set { _url = value; }
  }

  public int MembersCount {
    get { return _membersCount; } set { _membersCount = value; }
  }
}
```





13

Код VB .NET 9.0

```
Public Class User
    Private _name As String
    Private _admin As String
    Private _url As String
    Private _membersCount As Integer

    Public Property Name() As String
        Get
            Return _name
        End Get
        Set(ByVal value As String)
            _name = value
        End Set
    End Property

    Public Property Admin() As String
        Get
            Return _admin
        End Get
        Set(ByVal value As String)
            _admin = value
        End Set
    End Property

    Public Property URL() As String
        Get
            Return _url
        End Get
        Set(ByVal value As String)
            _url = value
        End Set
    End Property

    Public Property MembersCount() As Integer
        Get
            Return _membersCount
        End Get
        Set(ByVal value As Integer)
            _membersCount = value
        End Set
    End Property
End Class
```

Для того, чтобы объявить форум VBStreets, нам пришлось бы писать вот такое:

Код C# 3.0

```
User VBStreets = new User();
VBStreets.Admin = "RayShade";
VBStreets.MembersCount = 7436;
VBStreets.Name = "VBStreets";
```

```
VBStreets.URL = "http://www.VBStreets.ru";
```

Код VB .NET 9.0

```
Dim VBStreets As New User
VBStreets.Admin = "RayShade"
VBStreets.MembersCount = 7436
VBStreets.Name = "VBStreets"
VBStreets.URL = "http://www.VBStreets.ru"
```

К счастью, в C# 3.0 появилась такая вот конструкция:

Код C# 3.0

```
User VBStreets = new User(){Admin = "RayShade",
                            MembersCount = 7436,
                            Name = "VBStreets",
                            URL = "http://www.VBStreets.ru"};
```

Код VB .NET 9.0

```
Dim VBStreets As New User {Admin := "RayShade", _
                           MembersCount := 7436, _
                           Name := "VBStreets", _
                           URL := "http://www.VBStreets.ru"}
```

Честно говоря, мне не очень понравился синтаксис VB .NET с :=, уж больно это напоминает Паскаль... :)

Эта конструкция существенно упрощает составление запросов. Скажем, мы можем написать

Код C# 3.0

```
IEnumerable<User> makeMeSuperAdmin = users.Select(
                        user => new User {
                              Admin = "Exception",
                              MembersCount = user.MembersCount,
                              Name = user.Name,
                              URL = user.URL});
```

и даже

Код C# 3.0

```
string[] usernames = {«RayShade», «tyomitch», «GSerg»};
IEnumerable<User> myOwnUsers = usernames.Select(
                   username => new User {
                          Admin = "Exception",
                          MembersCount = 1000000,
                          Name = username,
                          URL = "http://" + username + ".ru"});
```

Ну а если писать, используя стандартную запись для операторов запросов, выйдет вот что:





15

Код C# 3.0

```
string[] usernames = { «RayShade», «tyomitch», «GSerg»};
IEnumerable<User> myOwnUsers = from username in usernames
                               select new User {
                                 Admin = "Exception",
                                 MembersCount = 1000000,
                                 Name = username,
                                 URL = "http://" + username + ".ru"};
```

Код VB .NET 9.0

```
Dim usernames() As String = {«RayShade», «tyomitch», «GSerg»}
Dim myOwnUsers As IEnumerable(Of User) = _
     Select New User { _
               Admin := "Exception", _
               MembersCount := 1000000, _
               Name := username, _
               URL := "http://" + username + ".ru"} _
     From username In usernames
```

Таким образом, мы получаем из массива `string` перечисление `IEnumerable<User>`.

## *Анонимные типы*

У Вас бывали ситуации, что нужно объявить класс только для одноразового использования? Нет? Значит, будут. Допустим, перед нами стоит задача вывести на консоль только данные о названии и количестве участников форума. Представим, что `User` – довольно большой класс, и использование его при выводе на консоль всего нескольких свойств может понизить производительность из-за большого расхода памяти. Какие у нас есть выходы?

1) *Без LINQ и анонимных типов*

   Рациональные методы отсутствуют (то есть без расхода памяти и с понятным кодом)

2) *С LINQ, без анонимных типов*

   Создать структуру `ForumStatistics` со свойствами `Username` и `MembersCount`. Затем делать выборку вот таким запросом:

Код C# 3.0

```
IEnumerable<User> users = //...
IEnumerable<ForumStatistics> forumStats = from user in users
                               select new ForumStatistics {
                                 MembersCount = user.MembersCount,
                                 Name = user.Name};
```

Код VB .NET 9.0

```
Dim users As IEnumerable(Of User) = '...
Dim forumStats As IEnumerable(Of ForumStatistics) = _
     Select New ForumStatistics { _
```

```
                MembersCount := user.MembersCount, _
                Name := user.Name} _
    From user In users
```

*3) С LINQ и анонимными типами*

Здесь всё ещё проще. Нам даже не нужно создавать структуру `ForumStatistics`. Компилятор создаст её автоматически, что сохранит нам время, а следовательно, деньги :)

Сейчас позволю себе небольшое лирическое отступление, т.к. не всё может быть понятно.

В C# 3.0 и VB .NET 9.0 можно писать код без объявления типов: компилятор сам поймёт, какой тип использовать. Например, код

Код C# 3.0

```
string s = "Bob";
int    n = 32;
bool   b = true;
```

Код VB .NET 9.0

```
Dim s As String = "Bob"
Dim n As Integer = 32
Dim b As Boolean = True
```

можно переписать, как

Код C# 3.0

```
var s = "Bob";
var n = 32;
var b = true;
```

Код VB .NET 9.0

```
Dim s = "Bob";
Dim n = 32
Dim b = True
```

, не теряя функциональности и производительности. С таким кодом даже работает IntelliSense! Правда, на такие объявления есть несколько ограничений:

i) Выражение всегда должно инициализироваться!

ii) Таким образом нельзя объявлять перечислимые типы

iii) Нельзя инициализировать выражение `null`'ем

Зачем же нужен этот «изврат», если можно так выразиться? Ответ прост: он позволяет не морочить себе голову при объявлении некоторых сложных типов и он также позволяет объявлять анонимные типы. Давайте рассмотрим наш пример. Код из пункта 2 можно переписать так:

Код C# 3.0





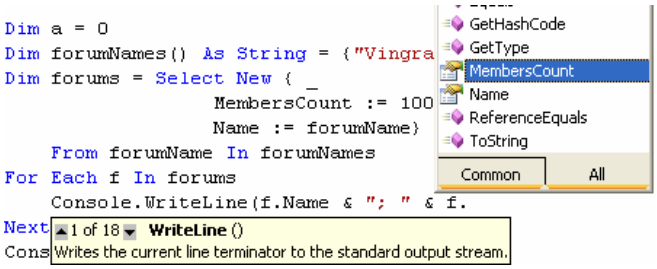
17

```
IEnumerable<User> users = //...
var forumStats = from user in users
                 select new {
                   MembersCount = user.MembersCount,
                   Name = user.Name};
```

Код VB .NET 9.0

```
Dim users As IEnumerable(Of User) = '...
Dim forumStats = _
     Select New { _
               MembersCount := user.MembersCount, _
               Name := user.Name} _
     From user In users
```

Самое приятное, что для анонимных типов по-прежнему доступен IntelliSense:




Впрочем, о поддержке IntelliSense, да и LINQ в VS 2005 мы поговорим позднее.

Если посмотреть Reflector'ом в сгенерированную сборку, видно, что компилятор сгенерировал анонимный тип `_Anonymous_Main_7_1C`. Кстати, использование анонимных типов имеет и обратную сторону: новички могут использовать их где попало, что, разумеется, затруднит чтение и понимание кода. Например, станет возможным творить вот такие «извраты»:

Код C# 3.0

```
var bob = new { Name = "Bob", Age = 51, CanCode = true };
var jane = new { Age = 29, FirstName = "Jane" };

var couple = new {
    Husband = new { bob.Name, bob.Age },
    Wife = new { Name = jane.FirstName, jane.Age }
};

int    ha = couple.Husband.Age; // ha == 51
string wn = couple.Wife.Name;   // wn == "Jane"
```

Код VB .NET 9.0

```
Dim bob = New {Name := "Bob", Age := 51, CanCode := True}
Dim jane = New {Age := 29, FirstName := "Jane"}
Dim couple = New { _
                 Husband := New {bob.Name, bob.Age}, _
                  Wife := New {Name := jane.FirstName, jane.Age} _
                 }
Dim ha As Integer = couple.Husband.Age ' ha = 51
Dim wn As String = couple.Wife.Name    ' wn = "Jane"
```

Кстати, свойства `Husband` и `Wife` получают имена `Name` и `Age` от соответствующих свойств `bob` и `jane`, им переданных. Можно написать и более подробно:

Код C# 3.0

```
var couple = new {
    Husband = new { Name = bob.Name, Age = bob.Age },
    Wife = new { Name = jane.FirstName, Age = jane.Age }
};
```

Код VB .NET 9.0

```
Dim couple = New { _
    Husband := New { Name := bob.Name, Age := bob.Age }, _
    Wife := new { Name := jane.FirstName, Age := jane.Age } _
}
```

## Остальные стандартные операторы

До этого момента мы рассматривали наиболее часто используемые операторы запросов: `Select` и `Where`. Сейчас мы приступим к рассмотрению остальных операторов запросов.

### *Сортировка и группировка*

Получить результаты запроса – это одно дело. Но зачастую требуется нечто большее, например бывает необходимо построить результаты запроса в определённом порядке. Здесь нам поможет оператор `OrderBy`.

Операторы `OrderBy` и `OrderByDescending` позволяют выстроить результирующие элементы в некотором заранее определённом порядке, например в порядке возрастания какого-нибудь свойства. Вот простенький пример их использования:

Код C# 3.0

```
IEnumerable<User> users = new User[]
{
     new User(){Admin = "RayShade",
           MembersCount = 7436,
           Name = "VBStreets",
           URL = "http://www.VBStreets.ru"},
      new User(){Admin = "Bill Gates",
           MembersCount = 12345,
           Name = "GotDotNet",
           URL = "http://gotdotnet.ru"},
```





19

```
        new User(){Admin = "IT",
             MembersCount = -1,
             Name = "RSDN",
             URL = "http://rsdn.ru"}
}
var sortedByMembers = users.OrderBy(f => f.MembersCount);
var sortedByName = users.OrderBy(f => f.Name);
var sortedByAdminDescending = users.OrderByDescending(f => f.Admin);
var sortedByNameLength = users.OrderBy(f => f.Name.Length);
```

Код VB .NET 9.0

```
Dim users As IEnumerable(Of User) = New User() { _
    New User {Admin := "RayShade", _
                MembersCount := 7436, _
                Name := "VBStreets", _
                URL := "http://www.VBStreets.ru"}, _
    New User {Admin := "Bill Gates", _
                MembersCount := 12345, _
                Name := "GotDotNet", _
                URL := "http://gotdotnet.ru"}, _
    New User {Admin := "IT", _
                MembersCount := -1, _
                Name := "RSDN", _
                URL := "http://rsdn.ru"} _
}
Dim sortedByMembers = Select aForum _
    From aForum In users _
    Order By aForum.MembersCount
Dim sortedByName = Select aForum _
    From aForum In users _
    Order By aForum.Name
Dim sortedByAdminDescending = Select aForum _
    From aForum In users _
    Order By aForum.Admin Descending
```

**Примечание**

Поскольку в VB .NET ещё не реализованы λ-выражения, примеров будет мало, а если и будут, то только написанные с использованием стандартного синтаксиса запросов, который поддерживается.

И оператор `OrderBy`, и `OrderByDescending` возвращают `SortedSequence<T>` для того, чтобы было возможным добавлять ещё критерии сортировки через операторы `ThenBy` и `ThenByDescending`:

Код C# 3.0

```
string[] someMembers = {"xolod", "RayShade", "tyomitch", "GSerg", "alibek", "Sebas"};
var bla = names.OrderBy(s => s.Length).ThenByDescending (s => s);
```

Код VB .NET 9.0

```
Dim someMembers As String() = _
     {"xolod", "RayShade", "tyomitch", "GSerg", "alibek", "Sebas"}
```

```
Dim bla = Select name _
    From name In someMembers _
    Order By name.Length, name Descending
```

Собственно говоря, это значит, что сначала все имена сортируются по количеству знаков, а те, у которых оно равное, сортируются по алфавиту:

```
"GSerg", "Sebas", "xolod", "alibek", "RayShade", "tyomitch"
```

Также к группе сортирующих операторов стоит добавить оператор `Reverse`, который попросту переставляет все элементы перечисления в обратном порядке.

Ещё предусмотрен оператор `GroupBy`, позволяющий группировать элементы перечисления по какому-либо признаку. Оператор `GroupBy` возвращает последовательность элементов `Grouping`, по одному для каждого уникального значения ключа. Впрочем, это проще показать на примере:

Код C# 3.0

```
string[] names = { "Albert", "Burke", "Connor", "David",
                   "Everett", "Frank", "George", "Harris"};

//группируем по количеству знаков
var grouping = names.GroupBy(s => s.Length);

foreach (Grouping<int, string> group in grouping) { //для каждой группы
    Console.WriteLine("Строки длины {0}", group.Key); //пишем кол-во символов

    foreach (string value in group.Group) //для каждого значения в группе
        Console.WriteLine("  {0}", value); //выписываем его
}
```

Этот код выводит на консоль

```
Строки длины 6
  Albert
  Connor
  George
  Harris
Строки длины 5
  Burke
  David
  Frank
Строки длины 7
  Everett
```

Сам интерфейс класса `Grouping` выглядит так:

Код C# 3.0





21

```
public sealed class Grouping<K, T> {
  public Grouping(K key, IEnumerable<T> group);
  public Grouping();
  public K Key { get; set; }
  public IEnumerable<T> Group { set; get; }
}
```

Код VB .NET 9.0

```
Public NotInheritable Class Grouping(Of K, T)
  Public Sub New (key As K, group As IEnumerable(Of T))
  Public Sub New()
  Public Property Key As K
  Public Property Group As IEnumerable(Of T)
End Class
```

Как и `Select`, оператор `GroupBy` позволяет указать функцию, показывающую, какие значения следует включить в группу:

Код C# 3.0

```
string[] names = { "Albert", "Burke", "Connor", "David",
                   "Everett", "Frank", "George", "Harris"};

var grouping = names.GroupBy(s => s.Length, //группируем по количеству знаков
                             s => s[0]);    //отбираем только первые символы
foreach (Grouping<int, char> group in grouping) {
    Console.WriteLine("Строки длины {0}", group.Key);

    foreach (char value in group.Group)
        Console.WriteLine("  {0}", value);
}
```

Этот код, как и следовало ожидать, выводит только первые символы:

```
Строки длины 6
  A
  C
  G
  H
Строки длины 5
  B
  D
  F
Строки длины 7
  E
```

## *Аггрегирующие операторы*

Наверняка, многие где-то слышали это слово. Как Вы помните, в SQL были выражения `COUNT`, `SUM` и т.п., которые возвращают не много строк, а только одно значение. Неудивительно, что эти операторы перекочевали и в LINQ.

Для начала мы рассмотрим новый оператор `Fold`. Вот как он выглядит:

Код C# 3.0

```
public static U Fold<T, U>(this IEnumerable<T> source,
                           U seed, Func<U, T, U> func) {
  U result = seed;

  foreach (T element in source)
      result = func(result, element);

  return result;
}
```

Наверное, это самый интересный аггрегирующий оператор. Суть его действия заключается в том, что при переборе он каждый раз выполняет λ-выражение, передавая в него предыдущий результат, так что получается эдакая «копилка». На самом деле это нетрудно понять из примера:

Код C# 3.0

```
string[] names = { "Albert", "Burke", "Connor", "David",
                   "Everett", "Frank", "George", "Harris"}; //имена

int count = names.Fold(0,                                //первоначальное значение
                       (counter, str) => counter + str.Length); //его изменение
// count == 46 (суммарное количество букв во всех словах)
```

В λ-выражение передаются два параметра:

- `counter` – *это текущее значение*
- `str` – *это текущий элемент*

Выражение возвращает результирующее значение `counter`, которое будет передано функции в следующий раз (или возвращено по окончании перебора).

Также, разумеется, определены операторы `Min`, `Max`, `Sum`, и `Average`, работающие только с числовыми типами. Тут всё предельно просто:

Код C# 3.0

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
string[] names = {"Albert", "Burke", "Connor", "David",
                  "Everett", "Frank", "George", "Harris"};

int totalNumbers = numbers.Sum();                  // totalNumbers == 55
int totalNamesLengths = names.Sum(s => s.Length); // totalNamesLengths == 46
```

Код VB .NET 9.0

```
Dim numbers As Integer() = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```





23

```
Dim names As String() = {"Albert", "Burke", "Connor", "David", _
                         "Everett", "Frank", "George", "Harris"}
Dim totalNumbers As Integer = numbers.Sum()
```

Кстати, подсчёт `totalNamesLengths` здесь аналогичен использованию `Fold` в предыдущем примере.

### *Select vs. SelectMany*

Оператор `Select` позволяет получать одно значение из исходного. Если необходимо получать несколько значений (к примеру, массив), необходимо прибегать к использованию промежуточной последовательности, что не очень удобно:

Код C# 3.0

```
string[] text = { "Albert was here",
                  "Burke slept late",
                  "Connor is happy" };

var tokens = text.Select(s => s.Split(' ')); //получаем последовательность массивов

foreach (string[] line in tokens)     //для каждого массива слов
    foreach (string token in line)    //для каждого слова в массиве
        Console.Write("{0}.", token); //вывести его с точкой на конце
```

Эта программа выводит на консоль следующий текст:

```
Albert.was.here.Burke.slept.late.Connor.is.happy.
```

В отличие от оператора `Select`, оператор `SelectMany` позволяет сразу запихнуть всё в одну последовательность:

```
string[] text = { "Albert was here",
                  "Burke slept late",
                  "Connor is happy" };

var tokens = text.SelectMany(s => s.Split(' ')); //собираем все слова в var

foreach (string token in tokens) //перечисляем их
    Console.Write("{0}.", token);
```

## Синтаксис запросов

Как я уже говорил, синтаксис запросов (*query syntax*) упрощает использование наиболее распространённых операторов запросов: `Where`, `Select`, `SelectMany`, `GroupBy`, `OrderBy`, `ThenBy`, `OrderByDescending`, и `ThenByDescending`.

Давайте вновь взглянем на приведённый мной пример в начале:

Код C# 3.0

```
IEnumerable<string> coolusers = users
                            .Where(f => f.Length == 7)
                            .OrderBy(f => f)
                            .Select(f => f.ToUpper());
```

С использованием синтаксиса запросов этот код выглядит так:

Код C# 3.0

```
IEnumerable<string> coolusers = from user in users
                                where user.Length == 7
                                orderby user
                                select user.ToUpper();
```

Код VB .NET 9.0

```
Dim coolusers As IEnumerable(Of String) = _
                                    Select user.ToUpper() _
                                     From user In users _
                                    Where user.Length = 7 _
                                    Order By user
```

Давайте взглянем на структуру выражения в C#. Выражение в C# начинается с первого выражения **`from...in`**, затем идут остальные выражения **`from...in`**, **`where`**, затем идёт выражение **`orderby`** с возможностью добавления ключевых слов **`ascending`** или **`descending`** (по умолчанию используется **`ascending`**). Ну и завершает наш список операторы **`select`** либо **`group...by`**.

Общая (правда, несколько упрощённая) схема выглядит примерно так:

```
from itemName in srcExpr
((from itemName in srcExpr) | (where predExpr))*
(orderby (keyExpr (ascending|descending)?)+)?
((select selExpr) | (group selExpr by keyExpr))
```

Вот пример двух выражений:

Код C# 3.0

```
var query1 = from p in people
             where p.Age > 20
             orderby p.Age descending, p.Name
             select new {
                 p.Name, Senior = p.Age > 30, p.CanCode
             };

var query2 = from p in people
             where p.Age > 20
             orderby p.Age descending, p.Name
             group new {
                p.Name, Senior = p.Age > 30, p.CanCode
             } by p.CanCode;
```

Компилятор преобразует их вот во что:





25

```
var query1 = people.Where(p => p.Age > 20)
                   .OrderByDescending(p => p.Age)
                   .ThenBy(p => p.Name)
                   .Select(p => new {
                       p.Name,
                       Senior = p.Age > 30,
                       p.CanCode
                   });

var query2 = people.Where(p => p.Age > 20)
                   .OrderByDescending(p => p.Age)
                   .ThenBy(p => p.Name)
                   .GroupBy(p => p.CanCode,
                            p => new {
                                   p.Name,
                                   Senior = p.Age > 30,
                                   p.CanCode
                   });
```

Мы также можем использовать сразу несколько источников отбора:

Код C# 3.0

```
var query = from s1 in names where s1.Length == 5
            from s2 in names where s1 == s2
            select s1 + " " + s2;
```

Код VB .NET 9.0

```
Dim query = Select It.s1 + " " + It.s2 _
From s1 In names, s2 In names _
Where It.s1.Length = 5 And It.s1 = It.s2
```

Как видно, синтаксис VB .NET несколько отличается: при объявлении нескольких источников данных в конструкции **`From...In`** необходимо обращаться к ним через ключевое слово `It`, а также все источники отбора разделять запятыми в одной конструкции **`From...In`** (в C# они просто перечисляются). Также в VB .NET все условия должны находиться в одонй конструкции **`Where`**.

Если запустить код, объявив массив `names` вот так:

Код C# 3.0

```
string[] names = {"Burke", "Connor", "Frank", "Everett",
                  "Albert", "George", "Harris", "David"};
```

Код VB .NET 9.0

```
Dim names() As String = {"Burke", "Connor", "Frank", "Everett", _
                         "Albert", "George",  "Harris", "David"}
```

, мы получим следующие результаты:

```
Burke Burke
Frank Frank
David David
```

Другая форма записи этого выражения выглядит так:

Код C# 3.0

```
var query = names.Where(s1 => s1.Length == 5)
                 .SelectMany(s1 =>
                     names.Where(s2 => s1 == s2)
                          .Select(s2 => s1 + " " + s2)
                 );
```

Заметьте, что благодаря использованию `SelectMany` мы сразу получаем строки, а не массивы строк.

Иногда бывает нужным проводить с группой, полученной с помощью **`group...by`**, некоторые дополнительные операции, например осуществить сортировку по значению ключа в обратном порядке. Чтобы сделать это, нам пришлось бы писать такой код:

Код C# 3.0

```
var query = from item in names
            orderby item.Length descending, item
            group item by item.Length;
```

Специально для таких целей существует ключевое слово **`into`**, позволяющее работать с группами, используя операторы **`orderby`** и **`select`**:

Код C# 3.0

```
var query = from item in names
            orderby item
            group item by item.Length into lengthGroups
            orderby lengthGroups.Key descending
            select lengthGroups;
```

Оба способа выводят одно и то же, но второй чуть более логичный (хотя и более длинный):

```
Strings of length 7
  Everett
Strings of length 6
  Albert
  Connor
  George
  Harris
Strings of length 5
  Burke
  David
  Frank
```

Только что мы обсудили, как C# и VB .NET реализуют так называемый *LINQ pattern*, способность языка предоставлять более удобные синтаксические конструкции для осуществления запросов. Сейчас мы вкратце рассмотрим два потомка LINQ – DLinq и XLinq, которые послужат темой моего следующего обзора.





27

## Интеграция с SQL

.NET Language Integrated Query позволяет использовать синтаксические конструкции, схожие с SQL, в обычных выражениях языка. Было бы крайне странно, если бы эта возможность не использовалась по своему прямому назначению – при работе с базами данных. Её плюсы очевидны: проверка типов при компиляции, IntelliSense... Сейчас мы рассмотрим, как же это реализуется.

DLinq определяет два главных атрибута, `[Table]` и `[Column]`, которые показывают нам, что соответствующий тип и его свойства (поля) отражают структуру базы данных.

Давайте вообразим, что форумы VBStreets, RSDN и GotDotNet решили объединиться и создать единую базу данных участников. Весьма упрощённо её структура выглядит так:

Код SQL

```
CREATE TABLE Users(
     [Name] nvarchar(30) NOT NULL,
     URL nvarchar(50) NOT NULL,
     ForumID uniqueidentifier ROWGUIDCOL NOT NULL,
)

CREATE TABLE Users(
     ForumID uniqueidentifier NOT NULL,
     [Login] nvarchar(15) NOT NULL,
     RealName nvarchar(50) NULL,
     RoleID uniqueidentifier NOT NULL,
     UserID uniqueidentifier ROWGUIDCOL NOT NULL
)


CREATE TABLE Roles(
     RoleID uniqueidentifier ROWGUIDCOL NOT NULL,
     RoleName nvarchar(15) NOT NULL
)
```

Соответствующие классы упрощённо (очень упрощённо :) ) выглядят так:

Код C# 3.0

```
[Table(Name = "Roles")]
public partial class Role
{

    [Column(DbType = "UniqueIdentifier NOT NULL", Id = true)]
    public System.Guid RoleID;

    [Column(DbType = "NVarChar(15) NOT NULL")]
    public string RoleName;
}
[Table(Name = "Users")]
public partial class User
{
    [Column(DbType = "UniqueIdentifier NOT NULL")]
    public System.Guid ForumID;

    [Column(DbType = "NVarChar(15) NOT NULL")]
    public string Login;
```

```
    [Column(DbType = "NVarChar(50)")]
    public string RealName;

    [Column(DbType = "UniqueIdentifier NOT NULL")]
    public System.Guid RoleID;

    [Column(DbType = "UniqueIdentifier NOT NULL", Id = true)]
    public System.Guid UserID;
}
[Table(Name = "Users")]
public partial class User
{
    [Column(DbType = "NVarChar(30) NOT NULL")]
    public string Name;

    [Column(DbType = "NVarChar(50) NOT NULL")]
    public string URL;

    [Column(DbType = "UniqueIdentifier NOT NULL", Id = true)]
    public System.Guid ForumID;
}
```

Код VB .NET 9.0

```
<Table(Name:="Roles")> _
Public Class Role

    <Column(DbType:="UniqueIdentifier NOT NULL", Id:=True)> _
    Public RoleID As System.Guid

    <Column(DbType:="NVarChar(15) NOT NULL")> _
    Public RoleName As String

End Class

<Table(Name:="Users")> _
Public Class User
    <Column(DbType:="UniqueIdentifier NOT NULL")> _
    Public ForumID As System.Guid

    <Column(DbType:="NVarChar(15) NOT NULL")> _
    Public Login As String

    <Column(DbType:="NVarChar(50)")> _
    Public RealName As String

    <Column(DbType:="UniqueIdentifier NOT NULL")> _
    Public RoleID As System.Guid
```



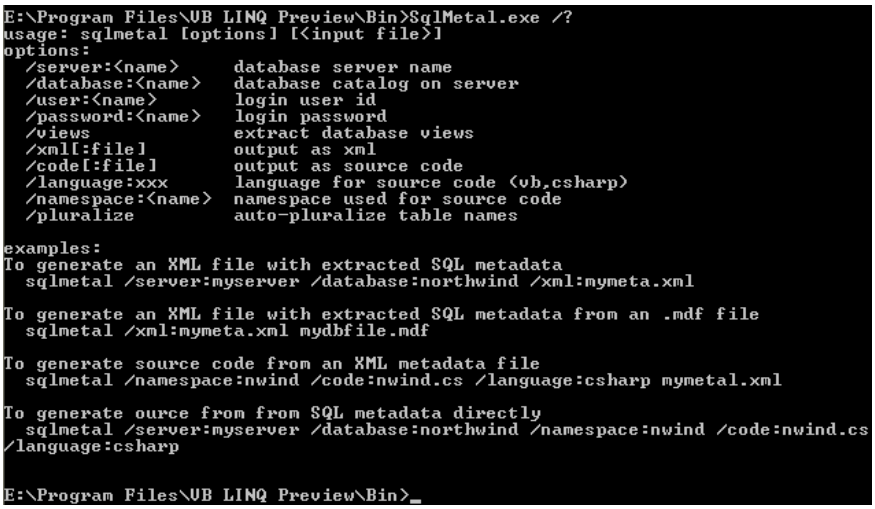


29

```
    <Column(DbType:="UniqueIdentifier NOT NULL", Id:=True)> _
    Public UserID As System.Guid

End Class

<Table(Name:="Users")> _
Public Class User

    <Column(DbType:="NVarChar(30) NOT NULL")> _
    Public Name As String

    <Column(DbType:="NVarChar(50) NOT NULL")> _
    Public URL As String

    <Column(DbType:="UniqueIdentifier NOT NULL", Id:=True)> _
    Public ForumID As System.Guid

End Class
```

Как видно, если тип CLR не полностью соответствует типу БД, он прописывается в `DbType`.

Собственно говоря, для того, чтобы преобразовать структуру БД в эквивалентный класс, лучше использовать специально созданную для этого утилиту SQLMetal (для SQL Server):




Мы её запустим с такими параметрами:

```
 SqlMetal /server:DANICH\SQL2005 /database:DLinqTest /user:sa /password:**********
/code:e:\dlinq.cs /language:csharp

 SqlMetal /server:DANICH\SQL2005 /database:DLinqTest /user:sa /password:**********
/code:e:\dlinq.vb /language:vb
```

В итоге у нас получатся два файла: `dlinq.vb` и `dlinq.cs`. К сожалению, детальное рассмотрение их содержания выходит за рамки данного обзора, но, будьте уверены, мы их рассмотрим в обзоре DLinq, который грядёт :)

Сейчас я кратко покажу их структуру, упустив многие важные, но более сложные конструкции.

Для начала рассмотрим простенький класс `DLinqTest`:

Код C# 3.0

```
public partial class DLinqTest : DataContext {

  public Table<Role> Roles;

  public Table<User> Users;

  public Table<User> Users;

  public DLinqTest() {
  }

  public DLinqTest(string connection) :
      base(connection) {
  }

  public DLinqTest(System.Data.IDbConnection connection) :
      base(connection) {
  }
}
```

Код VB .NET 9.0

```
Partial Public Class DLinqTest
    Inherits DataContext

    Public Roles As Table(Of Role)

    Public Users As Table(Of User)

    Public Users As Table(Of User)

    Public Sub New()
        MyBase.New()
    End Sub

    Public Sub New(ByVal connection As String)
        MyBase.New(connection)
    End Sub

    Public Sub New(ByVal connection As System.Data.IDbConnection)
        MyBase.New(connection)
    End Sub
End Class
```

Как видно, сначала определяется главный класс, наследующий от `DataContext`. Зачем он нужен?

Для того, чтобы получить таблицу Users, нам пришлось бы писать

Код C# 3.0





31

```
DataContext db = new DataContext(
     connection); //connection – раннее созданное соединение с БД
Table<User> users = db.GetTable<User>(); //получаем таблицу типу User
```

Код VB .NET 9.0

```
Dim db As New DataContext(connection) 'connection – раннее созданное соединение с БД
Dim users As Table(Of User) = db.GetTable(Of User)() 'получаем таблицу по типу User
```

Благодаря класу DLinqTest мы можем просто написать

Код C# 3.0

```
DLinqTest db = new DLinqTest (
     connection); //connection – раннее созданное соединение с БД
Table<User> users = db.Users; //берём таблицу из свойства
```

Код VB .NET 9.0

```
Dim db As New DLinqTest(connection) 'connection – раннее созданное соединение с БД
Dim users As Table(Of User) = db.Users 'берём таблицу из свойства
```

Вот пример кода, который работает с классами, сгенерированными SQLMetal:

Код C# 3.0

```
class Program
{
    static void Main()
    {
        SqlConnection connection = new SqlConnection
         (@"server=DANICH\SQL2005;uid=sa;pwd=**********;database=DLinqTest;");
        DLinqTest db = new DLinqTest(connection);
        var moderators = from u in db.Users
                         where u.Role.RoleName == "Moderator"
                         select new {Login = u.Login,
                                     RealName = u.RealName,
                                     Username = u.User.Name};
        foreach (var m in moderators)
            Console.WriteLine("{0}, {1}, {2};", m.Login, m.RealName, m.Username);
        Console.ReadKey();
    }
}
```

Код VB .NET 9.0

```
Module Program

    Sub Main()
        Dim connection As New SqlConnection _
        ("server=DANICH\SQL2005;uid=sa;pwd=**********;database=DLinqTest;")
        Dim db As New DLinqTest(connection)
        Dim moderators = Select New {Login := u.Login, _
```

```
                                        RealName := u.RealName, _
                                        Username := u.User.Name} _
            From u In db.Users _
            Where u.Role.RoleName = "Moderator"
        For Each m In moderators
            Console.WriteLine("{0}, {1}, {2};", m.Login, m.RealName, m.Username)
        Next
        Console.ReadKey()
    End Sub
End Module
```

Ещё один плюс такого подхода – поскольку `DataContext` в конструкторе принимает соединение в виде `IDbConnection`, то он может работать с любыми совместимыми с T-SQL базами данных.

Во время перебора элементов наш запрос транслируется из деревьев выражений в SQL-запрос.

## Интеграция с XML

Параллельно с DLinq разрабатывалась технология для доступа через запросы к данным в виде XML, которая была названа XLinq. Эта технология предоставляет доступ к таким функциям языка XML, как XPath, XQuery. XLinq призвана упростить доступ к информации в виде XML. В основном в XLinq используются три главных класса: `XName`, `XElement` и `XAttribute`, представляющие, соответственно, имя XML-узла, узел (node) XML-документа и атрибут узла.

Как можно заметить, конструктор `XElement` принимает два параметра – `XName` и «внутренности» элемента, выраженные `object`'ом. Обычно вместо `XName` передают просто строку с именем элемента, которая автоматически в него преобразуется. Далее можно передать любой объект (будет вызван метод `ToString()`), `Xelement` (дочерний узел) или `XAttribute` (он превратится в атрибут узла), причём в любом количестве, т.к. второй параметр объявлен как `ParamArray` aka `args`. Давайте взглянем на простенький примерчик.

Код C# 3.0

```
var xml = new XElement("User",
              new XAttribute("Name", "VBStreets"),
              new XAttribute("URL", "http://www.VBStreets.ru"),
              new XElement("Members",
                 new XElement("Member",
                     new XAttribute("Role","Moderator"),
                     "GSerg"),
```

```
                new XElement("Member",
                    new XAttribute("Role","Admin"),
                    "RayShade"),
                new XElement("Member",
                    new XAttribute("Role","Expert"),
                    "tyomitch")
            )
        );
Console.WriteLine(xml);
```

### Код VB .NET 9.0

```
Dim xml = New XElement("User", _
          New XAttribute("Name", "VBStreets"), _
          New XAttribute("URL", "http://www.VBStreets.ru"), _
          New XElement("Members", _
             New XElement("Member", _
                 New XAttribute("Role", "Moderator"), _
                 "GSerg"), _
             New XElement("Member", _
                 New XAttribute("Role", "Admin"), _
                 "RayShade"), _
             New XElement("Member", _
                 New XAttribute("Role", "Expert"), _
                 "tyomitch") _

          ))
Console.WriteLine(xml)
```

Этот код выводит вот что:

### Код XML

```
<User Name="VBStreets" URL="http://www.VBStreets.ru">
  <Members>
    <Member Role="Moderator">GSerg</Member>
    <Member Role="Admin">RayShade</Member>
    <Member Role="Expert">tyomitch</Member>
  </Members>
</User>
```

Кстати, в VB .NET 9.0 есть и другая возможность написания такого кода, называемая *XML literals*. Суть её сводится к тому, что XML-выражения интегрированы в язык и могут присваиваться переменным. На самом деле компилятор анализирует их и приводит к виду `XElement`. Например, пример выше мы можем переписать так:

### Код VB .NET 9.0

```
Dim xml = _
<User Name="VBStreets" URL="http://www.VBStreets.ru">
```

```
    <Members>
      <Member Role="Moderator">GSerg</Member>
      <Member Role="Admin">RayShade</Member>
      <Member Role="Expert">tyomitch</Member>
    </Members>
  </User>
```

Это выглядит нагляднее, не так ли? Поэтому в примерах для VB .NET 9.0 я буду использовать именно такой стиль. Кстати, переносы строк ( _ ) использовать в XML-литералах не нужно.

Мы также можем загрузить содержимое в `XElement` из файла, строки или `XmlReader`'а, но сейчас это нас не особо интересует.

Давайте рассмотрим простейший запрос.

### Код C# 3.0

```
var xmlItems = from user in users
           where user.URL == "http://www.VBStreets.ru"
           select new XElement("CoolusersAdmin",
                     new XAttribute("User",user.Name),
                     user.Admin
           );
foreach(var anXml in xmlItems) Console.WriteLine(anXml);
```

### Код VB .NET 9.0

```
Dim xmlItems = Select <CoolusersAdmin User=<%= User.Name %>>
                        <%= User.Admin %>
                      </CoolusersAdmin> _
    From user In users _
    Where User.URL = "http://www.VBStreets.ru"
For Each anXml In xmlItems
    Console.WriteLine(anXml)
Next
```

Этот код нам выдаст

### Код XML

```
<CoolForumAdmin User="VBStreets">RayShade</CoolForumAdmin>
```

Вместо перебора элементов `xmlItems` можно просто «обернуть» результат запроса:

### Код C# 3.0

```
var xmlItem = new XElement("Users",
        from user in users
           where user.URL == "http://www.VBStreets.ru"
           select new XElement("CoolusersAdmin",
                     new XAttribute("User",user.Name),
                     user.Admin
```





35

```
            ));
```

Код VB .NET 9.0

```
Dim xmlItem = <ForumInfo>
                  <%= Select <CoolusersAdmin User=<%= User.Name %>>
                                 <%=user.Admin %>
                             </CoolusersAdmin> _
                             From user In users _
                             Where User.URL = "http://www.VBStreets.ru" %>
              </ForumInfo>
```

Код XML

```
<ForumInfo>
    <CoolForumAdmin User="VBStreets">RayShade</CoolForumAdmin>
</ForumInfo>
```

Только что мы использовали запросы для составления XML-выражений. Их также можно использовать и для их «распаковки». Допустим, у нас есть следующая XML-структура:

Код XML

```
<ForumInfo>
    <User Name="VBStreets" URL="http://www.VBStreets.ru">
        <Admin>RayShade</Admin>
        <MembersCount>7436</MembersCount>
    </User>
</ForumInfo>
```

Следующий код создаст массив объектов `User` из неё:

Код C# 3.0

```
var xmlForumInfo = XElement.Parse(
@"<ForumInfo>
    <User Name=""VBStreets"" URL=""http://www.VBStreets.ru"">
        <Admin>RayShade</Admin>
        <MembersCount>7436</MembersCount>
    </User>
</ForumInfo>");
var user = from f in xmlForumInfo.Descendants("User")
           select new User {Name =(string)f.Attribute("Name"),
                            URL = (string)f.Attribute("URL"),
                            Admin = (string)f.Element("Admin"),
                            MembersCount = (int)f.Element("MembersCount")};
```

Код VB .NET 9.0

```
Dim xmlForumInfo = _
<ForumInfo>
    <User Name="VBStreets" URL="http://www.VBStreets.ru">
        <Admin>RayShade</Admin>
        <MembersCount>7436</MembersCount>
    </User>
</ForumInfo>
```

```
Dim user = Select New User {Name := f.@Name.Value, _
                            URL := f.@URL.Value, _
                            Admin := f.Admin(0).Value, _
                            MembersCount := f.MembersCount(0).Value} _
    From f In xmlForumInfo.User
```

VB .NET 9.0 опять-таки предоставляет более простой синтаксис с доступом к атрибутам через знак `@`, к дочерним элементам через простой синтаксис с точкой, а к элементам, расположенным «где-то глубоко» (дочерними дочерных и т.п. :) ) через три точки (или через метод `Descendants`). Минус такого подхода – приходится выключать `Option Strict`, что есть зло. К счастью, в недрах MS готовится некий аналог SQLMetal, который создаёт «типизированные» `XElement`'ы по схемам.

В C# же используются определённые `XAttribute`'ом и `XElement`'ом операторы приведения типа, которые возвращают нужное значение. Эти операторы определены для типов `string`, `bool`, `bool?`, `int`, `int?`, `uint`, `uint?`, `long`, `long?`, `ulong`, `ulong?`, `float`, `float?`, `double`, `double?`, `decimal`, `decimal?`, `DateTime`, `DateTime?`, `TimeSpan`, `TimeSpan?`, `GUID` и `GUID?`.

## Эпилог

Честно говоря, при переводе я сразу удалил эту часть статьи, ибо она была совсем неинтересная. Вместо этого, пожалуй, я напишу кое-что своё. Для начала хочу сказать, что при написании статьи использовалась статья The LINQ Project Overview (которая и послужила placeholder'ом для этой). Почти все примеры в статье мои (признаюсь, чуть-чуть было и простого перевода). Не знаю, какое впечатление у Вас сложилось о LINQ, но я надеюсь, что хорошее (это было целью написания статьи). Наверняка многие захотят поставить себе LINQ Preview прямо сейчас. Что для этого нужно сделать?

1. Зайдите на http://msdn.microsoft.com/vbasic/future или http://msdn.microsoft.com/vcsharp/future
2. Скачайте последний LINQ Technology Preview (под надписью Downloads)
3. Естественно, его надо установить :)
4. Если Вы ставите LINQ для C#, нужно ещё установить поддержку для VS 2005 (в VB она ставится автоматически), процедура установки описана в readme.html
5. Дерзайте!

На данный момент существует достаточно недоделок и исправлений, но всё это постоянно совершенствуется. Кстати, чтобы добавить функциональность LINQ, необходимо подключить библиотеки System.Query.dll (если надо - System.Xml.XLinq.dll и System.Data.DLinq.dll), которые лежат в LINQ Preview\Bin и добавить импорт на них. Также можно воспользоваться шаблоном LINQ Console/Windows Application.

Далее я приведу табличку со стандартными операторами запросов и их функциями:

| | |
|---|---|
| `OfType` | Filter based on type affiliation |
| `Select/SelectMany` | Project based on transform function |
| `Where` | Filter based on predicate function |
| `Count` | Count based on optional predicate function |
| `All/Any` | Universal/Existential quantification based on predicate function |
| `First/FirstOrDefault` | Access initial member based on optional predicate function |
| `ElementAt` | Access member at specified position |
| `Take/Skip` | Access members before/after specified position |
| `TakeWhile/SkipUntil` | Access members before/after predicate function is satisfied |
| `GroupBy` | Partition based on key extraction function |
| `ToDictionary` | Create key/value dictionary based on key extraction function |
| `OrderBy/ThenBy` | Sort in ascending order based on key extraction function and optional comparison function |
| `OrderByDescending/ ThenByDescending` | Sort in descending order based on key extraction function and optional comparison function |
| `Reverse` | Reverse the order of a sequence |
| `Fold` | Aggregate value over multiple values based on aggregation function |
| `Min/Max/Sum/Average` | Numeric aggregation functions |
| `Distinct` | Filter duplicate members |
| `Except` | Filter elements that are members of specified set |
| `Intersect` | Filter elements that are not members of specified set |
| `Union` | Combine distinct members from two sets |
| `Concat` | Concatenate the values of two sequences |
| `ToArray/ToList` | Buffer results of query in array or `List<T>` |
| `Range` | Create a sequence of numbers in a range |
| `Repeat` | Create a sequence of multiple copies of a given value |

И уж совсем напоследок хочу привести свободно распространяемый исходный код классов пространства имён System.Query. Попытайтесь в нём разобраться.

## Код C# 3.0

```
// Copyright (c) Microsoft Corporation.  All rights reserved.
using System;
using System.Collections;
using System.Collections.Generic;

namespace System.Query
{
    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, A2, A3, T>(A0 arg0, A1 arg1, A2 arg2, A3 arg3);

    public sealed class EmptySequenceException : Exception { }

    public static class Sequence
    {
        public static IEnumerable<T> Where<T>(this IEnumerable<T> source, Func<T, bool>
predicate) {
            foreach (T element in source) {
                if (predicate(element)) yield return element;
            }
        }

        public static IEnumerable<T> Where<T>(this IEnumerable<T> source, Func<T, int, bool>
predicate) {
            int index = 0;
            foreach (T element in source) {
                if (predicate(element, index)) yield return element;
                index++;
            }
        }

        public static IEnumerable<S> Select<T, S>(this IEnumerable<T> source, Func<T, S>
selector) {
            foreach (T element in source) {
                yield return selector(element);
            }
        }

        public static IEnumerable<S> Select<T, S>(this IEnumerable<T> source, Func<T, int, S>
selector) {
            int index = 0;
            foreach (T element in source) {
                yield return selector(element, index);
                index++;
            }
        }

        public static IEnumerable<S> SelectMany<T, S>(this IEnumerable<T> source, Func<T,
IEnumerable<S>> selector) {
            foreach (T element in source) {
                foreach (S subElement in selector(element)) {
```





39

```
                    yield return subElement;
                }
            }
        }

        public static IEnumerable<S> SelectMany<T, S>(this IEnumerable<T> source, Func<T, int,
IEnumerable<S>> selector) {
            int index = 0;
            foreach (T element in source) {
                foreach (S subElement in selector(element, index)) {
                    yield return subElement;
                }
                index++;
            }
        }

        public static IEnumerable<T> Take<T>(this IEnumerable<T> source, int count) {
            if (count > 0) {
                foreach (T element in source) {
                    yield return element;
                    if (--count == 0) break;
                }
            }
        }

        public static IEnumerable<T> TakeWhile<T>(this IEnumerable<T> source, Func<T, bool>
predicate) {
            foreach (T element in source) {
                if (!predicate(element)) break;
                yield return element;
            }
        }

        public static IEnumerable<T> TakeWhile<T>(this IEnumerable<T> source, Func<T, int,
bool> predicate) {
            int index = 0;
            foreach (T element in source) {
                if (!predicate(element, index)) break;
                yield return element;
                index++;
            }
        }

        public static IEnumerable<T> Skip<T>(this IEnumerable<T> source, int count) {
            using (IEnumerator<T> e = source.GetEnumerator()) {
                while (count > 0 && e.MoveNext()) count--;
                if (count <= 0) {
                    while (e.MoveNext()) yield return e.Current;
                }
            }
        }

        public static IEnumerable<T> SkipWhile<T>(this IEnumerable<T> source, Func<T, bool>
predicate) {
            bool yielding = false;
            foreach (T element in source) {
```

```
                if (!yielding && !predicate(element)) yielding = true;
                if (yielding) yield return element;
            }
        }

        public static IEnumerable<T> SkipWhile<T>(this IEnumerable<T> source, Func<T, int,
bool> predicate) {
            int index = 0;
            bool yielding = false;
            foreach (T element in source) {
                if (!yielding && !predicate(element, index)) yielding = true;
                if (yielding) yield return element;
                index++;
            }
        }

        public static OrderedSequence<T> OrderBy<T, K>(this IEnumerable<T> source, Func<T, K>
keySelector) {
            return new OrderedSequence<T, K>(source, null, keySelector, null, false);
        }

        public static OrderedSequence<T> OrderBy<T, K>(this IEnumerable<T> source, Func<T, K>
keySelector, IComparer<K> comparer) {
            return new OrderedSequence<T, K>(source, null, keySelector, comparer, false);
        }

        public static OrderedSequence<T> OrderByDescending<T, K>(this IEnumerable<T> source,
Func<T, K> keySelector) {
            return new OrderedSequence<T, K>(source, null, keySelector, null, true);
        }

        public static OrderedSequence<T> OrderByDescending<T, K>(this IEnumerable<T> source,
Func<T, K> keySelector, IComparer<K> comparer) {
            return new OrderedSequence<T, K>(source, null, keySelector, comparer, true);
        }

        public static OrderedSequence<T> ThenBy<T, K>(this OrderedSequence<T> source, Func<T,
K> keySelector) {
            return new OrderedSequence<T, K>(source.source, source, keySelector, null, false);
        }

        public static OrderedSequence<T> ThenBy<T, K>(this OrderedSequence<T> source, Func<T,
K> keySelector, IComparer<K> comparer) {
            return new OrderedSequence<T, K>(source.source, source, keySelector, comparer,
false);
        }

        public static OrderedSequence<T> ThenByDescending<T, K>(this OrderedSequence<T>
source, Func<T, K> keySelector) {
            return new OrderedSequence<T, K>(source.source, source, keySelector, null, true);
        }

        public static OrderedSequence<T> ThenByDescending<T, K>(this OrderedSequence<T>
source, Func<T, K> keySelector, IComparer<K> comparer) {
```





41

```
            return new OrderedSequence<T, K>(source.source, source, keySelector, comparer,
true);
        }

        public static IEnumerable<Grouping<K, T>> GroupBy<T, K>(this IEnumerable<T> source,
Func<T, K> keySelector) {
            return GroupBy(source, keySelector, null);
        }

        public static IEnumerable<Grouping<K, T>> GroupBy<T, K>(this IEnumerable<T> source,
Func<T, K> keySelector, IEqualityComparer<K> comparer) {
            Dictionary<K, List<T>> dict = new Dictionary<K, List<T>>(comparer);
            foreach (T element in source) {
                K key = keySelector(element);
                List<T> list;
                if (!dict.TryGetValue(key, out list)) {
                    list = new List<T>();
                    dict.Add(key, list);
                }
                list.Add(element);
            }
            foreach (KeyValuePair<K, List<T>> pair in dict) {
                yield return new Grouping<K, T>(pair.Key, pair.Value);
            }
        }

        public static IEnumerable<Grouping<K, E>> GroupBy<T, K, E>(this IEnumerable<T> source,
Func<T, K> keySelector, Func<T, E> elemSelector) {
            return GroupBy(source, keySelector, elemSelector, null);
        }

        public static IEnumerable<Grouping<K, E>> GroupBy<T, K, E>(this IEnumerable<T> source,
Func<T, K> keySelector, Func<T,E> elemSelector, IEqualityComparer<K> comparer) {
            Dictionary<K, List<E>> dict = new Dictionary<K, List<E>>(comparer);
            foreach (T element in source) {
                K key = keySelector(element);
                E elem = elemSelector(element);
                List<E> list;
                if (!dict.TryGetValue(key, out list)) {
                    list = new List<E>();
                    dict.Add(key, list);
                }
                list.Add(elem);
            }
            foreach (KeyValuePair<K, List<E>> pair in dict) {
                yield return new Grouping<K, E>(pair.Key, pair.Value);
            }
        }

        public static IEnumerable<T> Concat<T>(this IEnumerable<T> first, IEnumerable<T>
second) {
            foreach (T element in first) yield return element;
            foreach (T element in second) yield return element;
        }

        public static IEnumerable<T> Distinct<T>(this IEnumerable<T> source) {
```

```
            Dictionary<T, object> dict = new Dictionary<T, object>();
            foreach (T element in source) {
                if (!dict.ContainsKey(element)) {
                    dict.Add(element, null);
                    yield return element;
                }
            }
        }

        public static IEnumerable<T> Union<T>(this IEnumerable<T> first, IEnumerable<T>
second) {
            Dictionary<T, object> dict = new Dictionary<T, object>();
            foreach (T element in first) {
                if (!dict.ContainsKey(element)) {
                    dict.Add(element, null);
                    yield return element;
                }
            }
            foreach (T element in second) {
                if (!dict.ContainsKey(element)) {
                    dict.Add(element, null);
                    yield return element;
                }
            }
        }

        public static IEnumerable<T> Intersect<T>(this IEnumerable<T> first, IEnumerable<T>
second) {
            Dictionary<T, object> dict = new Dictionary<T, object>();
            foreach (T element in first) dict[element] = null;
            foreach (T element in second) {
                if (dict.ContainsKey(element)) dict[element] = dict;
            }
            foreach (KeyValuePair<T, object> pair in dict) {
                if (pair.Value != null) yield return pair.Key;
            }
        }

        public static IEnumerable<T> Except<T>(this IEnumerable<T> first, IEnumerable<T>
second) {
            Dictionary<T, object> dict = new Dictionary<T, object>();
            foreach (T element in first) dict[element] = null;
            foreach (T element in second) dict.Remove(element);
            foreach (T element in dict.Keys) yield return element;
        }

        public static IEnumerable<T> Reverse<T>(this IEnumerable<T> source) {
            Buffer<T> buffer = new Buffer<T>(source);
            for (int i = buffer.count - 1; i >= 0; i--) yield return buffer.items[i];
        }

        public static bool EqualAll<T>(this IEnumerable<T> first, IEnumerable<T> second) {
            using (IEnumerator<T> e1 = first.GetEnumerator())
                using (IEnumerator<T> e2 = second.GetEnumerator()) {
                    while (e1.MoveNext()) {
```





43

```
                        if (!(e2.MoveNext() && Equals(e1.Current, e2.Current))) return false;
                    }
                    if (e2.MoveNext()) return false;
                }
            return true;
        }

        public static T[] ToArray<T>(this IEnumerable<T> source) {
            return new Buffer<T>(source).ToArray();
        }

        public static List<T> ToList<T>(this IEnumerable<T> source) {
            return new List<T>(source);
        }

        public static Dictionary<K, T> ToDictionary<T, K>(this IEnumerable<T> source, Func<T,
K> keySelector) {
            Dictionary<K, T> d = new Dictionary<K, T>();
            foreach (T element in source) d.Add(keySelector(element), element);
            return d;
        }

        public static Dictionary<K, E> ToDictionary<T, K, E>(this IEnumerable<T> source,
Func<T, K> keySelector, Func<T, E> elemSelector) {
            Dictionary<K, E> d = new Dictionary<K, E>();
            foreach (T element in source) d.Add(keySelector(element), elemSelector(element));
            return d;
        }

        public static IEnumerable<T> OfType<T>(this IEnumerable source) {
            foreach (object obj in source) {
                if (obj is T) yield return (T)obj;
            }
        }

        public static IEnumerable<T> ToSequence<T>(this IEnumerable<T> seq) {
            return seq;
        }

        public static T First<T>(this IEnumerable<T> source) {
            foreach (T element in source) {
                return element;
            }
            throw new EmptySequenceException();
        }

        public static T First<T>(this IEnumerable<T> source, Func<T, bool> predicate) {
            foreach (T element in source) {
                if (predicate(element)) return element;
            }
            throw new EmptySequenceException();
        }

        public static T First<T>(this IEnumerable<T> source, Func<T, int, bool> predicate) {
            int index = 0;
            foreach (T element in source) {
```

```
                if (predicate(element, index)) return element;
                index++;
            }
            throw new EmptySequenceException();
        }

        public static T FirstOrDefault<T>(this IEnumerable<T> source) {
            foreach (T element in source) {
                return element;
            }
            return default(T);
        }

        public static T FirstOrDefault<T>(this IEnumerable<T> source, Func<T, bool> predicate)
{
            foreach (T element in source) {
                if (predicate(element)) return element;
            }
            return default(T);
        }

        public static T FirstOrDefault<T>(this IEnumerable<T> source, Func<T, int, bool>
predicate) {
            int index = 0;
            foreach (T element in source) {
                if (predicate(element, index)) return element;
                index++;
            }
            return default(T);
        }

        public static T ElementAt<T>(this IEnumerable<T> source, int index) {
            IList<T> list = source as IList<T>;
            if (list != null) return list[index];
            if (index < 0) throw new ArgumentException();
            using (IEnumerator<T> e = source.GetEnumerator()) {
                while (true) {
                    if (!e.MoveNext()) throw new ArgumentException();
                    if (index == 0) return e.Current;
                    index--;
                }
            }
        }

        public static IEnumerable<int> Range(int start, int count) {
            if (count < 0) throw new ArgumentException();
            for (int i = 0; i < count; i++) yield return start + i;
        }

        public static IEnumerable<T> Repeat<T>(T element, int count) {
            if (count < 0) throw new ArgumentException();
            for (int i = 0; i < count; i++) yield return element;
        }

        public static bool Any<T>(this IEnumerable<T> source) {
```





45

```
        foreach (T element in source) {
            return true;
        }
        return false;
    }

    public static bool Any<T>(this IEnumerable<T> source, Func<T, bool> predicate) {
        foreach (T element in source) {
            if (predicate(element)) return true;
        }
        return false;
    }

    public static bool Any<T>(this IEnumerable<T> source, Func<T, int, bool> predicate) {
        int index = 0;
        foreach (T element in source) {
            if (predicate(element, index)) return true;
            index++;
        }
        return false;
    }

    public static bool All<T>(this IEnumerable<T> source, Func<T, bool> predicate) {
        foreach (T element in source) {
            if (!predicate(element)) return false;
        }
        return true;
    }

    public static bool All<T>(this IEnumerable<T> source, Func<T, int, bool> predicate) {
        int index = 0;
        foreach (T element in source) {
            if (!predicate(element, index)) return false;
            index++;
        }
        return true;
    }

    public static int Count<T>(this IEnumerable<T> source) {
        ICollection<T> collection = source as ICollection<T>;
        if (collection != null) return collection.Count;
        int count = 0;
        using (IEnumerator<T> e = source.GetEnumerator()) {
            checked {
                while (e.MoveNext()) count++;
            }
        }
        return count;
    }

    public static int Count<T>(this IEnumerable<T> source, Func<T, bool> predicate) {
        int count = 0;
        foreach (T element in source) {
            checked {
                if (predicate(element)) count++;
            }
```

```
        }
        return count;
    }

    public static int Count<T>(this IEnumerable<T> source, Func<T, int, bool> predicate) {
        int count = 0;
        int index = 0;
        foreach (T element in source) {
            checked {
                if (predicate(element, index)) count++;
            }
            index++;
        }
        return count;
    }

    public static bool Contains<T>(this IEnumerable<T> source, T value) {
        if (value == null) {
            foreach (T element in source)
                if (element == null) return true;
        }
        else {
            EqualityComparer<T> c = EqualityComparer<T>.Default;
            foreach (T element in source)
                if (c.Equals(element, value)) return true;
        }
        return false;
    }

    public static T Fold<T>(this IEnumerable<T> source, Func<T, T, T> func) {
        using (IEnumerator<T> e = source.GetEnumerator()) {
            if (!e.MoveNext()) throw new EmptySequenceException();
            T result = e.Current;
            while (e.MoveNext()) result = func(result, e.Current);
            return result;
        }
    }

    public static U Fold<T, U>(this IEnumerable<T> source, U seed, Func<U, T, U> func) {
        U result = seed;
        foreach (T element in source) result = func(result, element);
        return result;
    }

    public static int Sum(this IEnumerable<int> source) {
        int sum = 0;
        checked {
            foreach (int v in source) sum += v;
        }
        return sum;
    }

    public static int? Sum(this IEnumerable<int?> source) {
        int? sum = 0;
        checked {
```





47

```
            foreach (int? v in source) {
                if (v != null) sum += v;
            }
        }
        return sum;
    }

    public static long Sum(this IEnumerable<long> source) {
        long sum = 0;
        checked {
            foreach (long v in source) sum += v;
        }
        return sum;
    }

    public static long? Sum(this IEnumerable<long?> source) {
        long? sum = 0;
        checked {
            foreach (long? v in source) {
                if (v != null) sum += v;
            }
        }
        return sum;
    }

    public static double Sum(this IEnumerable<double> source) {
        double sum = 0;
        foreach (double v in source) sum += v;
        return sum;
    }

    public static double? Sum(this IEnumerable<double?> source) {
        double? sum = 0;
        foreach (double? v in source) {
            if (v != null) sum += v;
        }
        return sum;
    }

    public static decimal Sum(this IEnumerable<decimal> source) {
        decimal sum = 0;
        foreach (decimal v in source) sum += v;
        return sum;
    }

    public static decimal? Sum(this IEnumerable<decimal?> source) {
        decimal? sum = 0;
        foreach (decimal? v in source) {
            if (v != null) sum += v;
        }
        return sum;
    }

    public static int Sum<T>(this IEnumerable<T> source, Func<T, int> expr) {
        return Sequence.Sum(Sequence.Select(source, expr));
    }
```

```
public static int? Sum<T>(this IEnumerable<T> source, Func<T, int?> expr) {
    return Sequence.Sum(Sequence.Select(source, expr));
}

public static double Sum<T>(this IEnumerable<T> source, Func<T, double> expr) {
    return Sequence.Sum(Sequence.Select(source, expr));
}

public static double? Sum<T>(this IEnumerable<T> source, Func<T, double?> expr) {
    return Sequence.Sum(Sequence.Select(source, expr));
}

public static long Sum<T>(this IEnumerable<T> source, Func<T, long> expr) {
    return Sequence.Sum(Sequence.Select(source, expr));
}

public static long? Sum<T>(this IEnumerable<T> source, Func<T, long?> expr) {
    return Sequence.Sum(Sequence.Select(source, expr));
}

public static decimal Sum<T>(this IEnumerable<T> source, Func<T, decimal> expr) {
    return Sequence.Sum(Sequence.Select(source, expr));
}

public static decimal? Sum<T>(this IEnumerable<T> source, Func<T, decimal?> expr) {
    return Sequence.Sum(Sequence.Select(source, expr));
}

public static int Min(this IEnumerable<int> source) {
    int value = 0;
    bool hasValue = false;
    foreach (int x in source) {
        if (hasValue) {
            if (x < value) value = x;
        }
        else {
            value = x;
            hasValue = true;
        }
    }
    if (hasValue) return value;
    throw new EmptySequenceException();
}

public static int? Min(this IEnumerable<int?> source) {
    int? value = null;
    foreach (int? x in source) {
        if (value == null || x < value) value = x;
    }
    return value;
}

public static long Min(this IEnumerable<long> source) {
    long value = 0;
```





49

```
        bool hasValue = false;
        foreach (long x in source) {
            if (hasValue) {
                if (x < value) value = x;
            }
            else {
                value = x;
                hasValue = true;
            }
        }
        if (hasValue) return value;
        throw new EmptySequenceException();
    }

    public static long? Min(this IEnumerable<long?> source) {
        long? value = null;
        foreach (long? x in source) {
            if (value == null || x < value) value = x;
        }
        return value;
    }

    public static double Min(this IEnumerable<double> source) {
        double value = 0;
        bool hasValue = false;
        foreach (double x in source) {
            if (hasValue) {
                if (x < value) value = x;
            }
            else {
                value = x;
                hasValue = true;
            }
        }
        if (hasValue) return value;
        throw new EmptySequenceException();
    }

    public static double? Min(this IEnumerable<double?> source) {
        double? value = null;
        foreach (double? x in source) {
            if (value == null || x < value) value = x;
        }
        return value;
    }

    public static decimal Min(this IEnumerable<decimal> source) {
        decimal value = 0;
        bool hasValue = false;
        foreach (decimal x in source) {
            if (hasValue) {
                if (x < value) value = x;
            }
            else {
                value = x;
                hasValue = true;
```

```
            }
        }
        if (hasValue) return value;
        throw new EmptySequenceException();
    }

    public static decimal? Min(this IEnumerable<decimal?> source) {
        decimal? value = null;
        foreach (decimal? x in source) {
            if (value == null || x < value) value = x;
        }
        return value;
    }

    public static T Min<T>(this IEnumerable<T> source) {
        Comparer<T> comparer = Comparer<T>.Default;
        T value = default(T);
        bool hasValue = false;
        foreach (T x in source) {
            if (hasValue) {
                if (comparer.Compare(x, value) < 0)
                    value = x;
            }
            else {
                value = x;
                hasValue = true;
            }
        }
        if (hasValue) return value;
        throw new EmptySequenceException();
    }

    public static int Min<T>(this IEnumerable<T> source, Func<T, int> map) {
        return Sequence.Min(Sequence.Select(source, map));
    }

    public static int? Min<T>(this IEnumerable<T> source, Func<T, int?> map) {
        return Sequence.Min(Sequence.Select(source, map));
    }

    public static long Min<T>(this IEnumerable<T> source, Func<T, long> map) {
        return Sequence.Min(Sequence.Select(source, map));
    }

    public static long? Min<T>(this IEnumerable<T> source, Func<T, long?> map) {
        return Sequence.Min(Sequence.Select(source, map));
    }

    public static double Min<T>(this IEnumerable<T> source, Func<T, double> map) {
        return Sequence.Min(Sequence.Select(source, map));
    }

    public static double? Min<T>(this IEnumerable<T> source, Func<T, double?> map) {
        return Sequence.Min(Sequence.Select(source, map));
    }
```





51

```
public static decimal Min<T>(this IEnumerable<T> source, Func<T, decimal> map) {
    return Sequence.Min(Sequence.Select(source, map));
}

public static decimal? Min<T>(this IEnumerable<T> source, Func<T, decimal?> map) {
    return Sequence.Min(Sequence.Select(source, map));
}

public static S Min<T, S>(this IEnumerable<T> source, Func<T, S> map) {
    return Sequence.Min(Sequence.Select(source, map));
}

public static int Max(this IEnumerable<int> source) {
    int value = 0;
    bool hasValue = false;
    foreach (int x in source) {
        if (hasValue) {
            if (x > value) value = x;
        }
        else {
            value = x;
            hasValue = true;
        }
    }
    if (hasValue) return value;
    throw new EmptySequenceException();
}

public static int? Max(this IEnumerable<int?> source) {
    int? value = null;
    foreach (int? x in source) {
        if (value == null || x > value) value = x;
    }
    return value;
}

public static long Max(this IEnumerable<long> source) {
    long value = 0;
    bool hasValue = false;
    foreach (long x in source) {
        if (hasValue) {
            if (x > value) value = x;
        }
        else {
            value = x;
            hasValue = true;
        }
    }
    if (hasValue) return value;
    throw new EmptySequenceException();
}

public static long? Max(this IEnumerable<long?> source) {
    long? value = null;
    foreach (long? x in source) {
```

```
            if (value == null || x > value) value = x;
        }
        return value;
    }

    public static double Max(this IEnumerable<double> source) {
        double value = 0;
        bool hasValue = false;
        foreach (double x in source) {
            if (hasValue) {
                if (x > value) value = x;
            }
            else {
                value = x;
                hasValue = true;
            }
        }
        if (hasValue) return value;
        throw new EmptySequenceException();
    }

    public static double? Max(this IEnumerable<double?> source) {
        double? value = null;
        foreach (double? x in source) {
            if (value == null || x > value) value = x;
        }
        return value;
    }

    public static decimal Max(this IEnumerable<decimal> source) {
        decimal value = 0;
        bool hasValue = false;
        foreach (decimal x in source) {
            if (hasValue) {
                if (x > value) value = x;
            }
            else {
                value = x;
                hasValue = true;
            }
        }
        if (hasValue) return value;
        throw new EmptySequenceException();
    }

    public static decimal? Max(this IEnumerable<decimal?> source) {
        decimal? value = null;
        foreach (decimal? x in source) {
            if (value == null || x > value) value = x;
        }
        return value;
    }

    public static T Max<T>(this IEnumerable<T> source) {
        Comparer<T> comparer = Comparer<T>.Default;
```





53

```
            T value = default(T);
            bool hasValue = false;
            foreach (T x in source) {
                if (hasValue) {
                    if (comparer.Compare(x, value) > 0)
                        value = x;
                }
                else {
                    value = x;
                    hasValue = true;
                }
            }
            if (hasValue) return value;
            throw new EmptySequenceException();
        }

        public static int Max<T>(this IEnumerable<T> source, Func<T, int> map) {
            return Sequence.Max(Sequence.Select(source, map));
        }

        public static int? Max<T>(this IEnumerable<T> source, Func<T, int?> map) {
            return Sequence.Max(Sequence.Select(source, map));
        }

        public static long Max<T>(this IEnumerable<T> source, Func<T, long> map) {
            return Sequence.Max(Sequence.Select(source, map));
        }

        public static long? Max<T>(this IEnumerable<T> source, Func<T, long?> map) {
            return Sequence.Max(Sequence.Select(source, map));
        }

        public static double Max<T>(this IEnumerable<T> source, Func<T, double> map) {
            return Sequence.Max(Sequence.Select(source, map));
        }

        public static double? Max<T>(this IEnumerable<T> source, Func<T, double?> map) {
            return Sequence.Max(Sequence.Select(source, map));
        }

        public static decimal Max<T>(this IEnumerable<T> source, Func<T, decimal> map) {
            return Sequence.Max(Sequence.Select(source, map));
        }

        public static decimal? Max<T>(this IEnumerable<T> source, Func<T, decimal?> map) {
            return Sequence.Max(Sequence.Select(source, map));
        }

        public static S Max<T, S>(this IEnumerable<T> source, Func<T, S> map) {
            return Sequence.Max(Sequence.Select(source, map));
        }

        public static double Average(this IEnumerable<int> source) {
            long sum = 0;
            int count = 0;
            checked {
```

```
            foreach (int v in source) {
                sum += v;
                count++;
            }
        }
        if (count > 0) return (double)sum / count;
        throw new EmptySequenceException();
    }

    public static double? Average(this IEnumerable<int?> source) {
        long sum = 0;
        int count = 0;
        checked {
            foreach (int? v in source) {
                if (v != null) {
                    sum += v.GetValueOrDefault();
                    count++;
                }
            }
        }
        if (count > 0) return (double)sum / count;
        return null;
    }

    public static double Average(this IEnumerable<long> source) {
        long sum = 0;
        int count = 0;
        checked {
            foreach (long v in source) {
                sum += v;
                count++;
            }
        }
        if (count > 0) return (double)sum / count;
        throw new EmptySequenceException();
    }

    public static double? Average(this IEnumerable<long?> source) {
        long sum = 0;
        long count = 0;
        checked {
            foreach (long? v in source) {
                if (v != null) {
                    sum += v.GetValueOrDefault();
                    count++;
                }
            }
        }
        if (count > 0) return (double)sum / count;
        return null;
    }

    public static double Average(this IEnumerable<double> source) {
        double sum = 0;
        int count = 0;
```





55

```
        checked {
            foreach (double v in source) {
                sum += v;
                count++;
            }
        }
        if (count > 0) return sum / count;
        throw new EmptySequenceException();
    }

    public static double? Average(this IEnumerable<double?> source) {
        double sum = 0;
        int count = 0;
        checked {
            foreach (double? v in source) {
                if (v != null) {
                    sum += v.GetValueOrDefault();
                    count++;
                }
            }
        }
        if (count > 0) return sum / count;
        return null;
    }

    public static decimal Average(this IEnumerable<decimal> source) {
        decimal sum = 0;
        int count = 0;
        checked {
            foreach (decimal v in source) {
                sum += v;
                count++;
            }
        }
        if (count > 0) return sum / count;
        throw new EmptySequenceException();
    }

    public static decimal? Average(this IEnumerable<decimal?> source) {
        decimal sum = 0;
        int count = 0;
        checked {
            foreach (decimal? v in source) {
                if (v != null) {
                    sum += v.GetValueOrDefault();
                    count++;
                }
            }
        }
        if (count > 0) return sum / count;
        throw new EmptySequenceException();
    }

    public static double Average<T>(this IEnumerable<T> source, Func<T, int> map) {
        return Sequence.Average(Sequence.Select(source, map));
    }
```

```
        public static double? Average<T>(this IEnumerable<T> source, Func<T, int?> map) {
            return Sequence.Average(Sequence.Select(source, map));
        }

        public static double Average<T>(this IEnumerable<T> source, Func<T, long> map) {
            return Sequence.Average(Sequence.Select(source, map));
        }

        public static double? Average<T>(this IEnumerable<T> source, Func<T, long?> map) {
            return Sequence.Average(Sequence.Select(source, map));
        }

        public static double Average<T>(this IEnumerable<T> source, Func<T, double> map) {
            return Sequence.Average(Sequence.Select(source, map));
        }

        public static double? Average<T>(this IEnumerable<T> source, Func<T, double?> map) {
            return Sequence.Average(Sequence.Select(source, map));
        }

        public static decimal Average<T>(this IEnumerable<T> source, Func<T, decimal> map) {
            return Sequence.Average(Sequence.Select(source, map));
        }

        public static decimal? Average<T>(this IEnumerable<T> source, Func<T, decimal?> map) {
            return Sequence.Average(Sequence.Select(source, map));
        }
    }

    public sealed class Grouping<K, T>
    {
        K key;
        IEnumerable<T> group;

        public Grouping(K key, IEnumerable<T> group) {
            this.key = key;
            this.group = group;
        }

        public Grouping() {
        }

        public K Key {
            get { return key; }
            set { key = value; }
        }

        public IEnumerable<T> Group {
            get { return group; }
            set { group = value; }
        }
    }

    public abstract class OrderedSequence<T>: IEnumerable<T>
```





57

```
    {
        internal IEnumerable<T> source;

        public IEnumerator<T> GetEnumerator() {
            Buffer<T> buffer = new Buffer<T>(source);
            if (buffer.count > 0) {
                SequenceSorter<T> sorter = GetSequenceSorter(null);
                int[] map = sorter.Sort(buffer.items, buffer.count);
                sorter = null;
                for (int i = 0; i < buffer.count; i++) yield return buffer.items[map[i]];
            }
        }

        internal virtual SequenceSorter<T> GetSequenceSorter(SequenceSorter<T> next) {
            throw new NotImplementedException();
        }

        IEnumerator IEnumerable.GetEnumerator() {
            return GetEnumerator();
        }
    }

    internal class OrderedSequence<T, K>: OrderedSequence<T>
    {
        internal OrderedSequence<T> parent;
        internal Func<T, K> keySelector;
        internal IComparer<K> comparer;
        internal bool descending;

        internal OrderedSequence(IEnumerable<T> source, OrderedSequence<T> parent, Func<T, K>
keySelector, IComparer<K> comparer, bool descending) {
            this.source = source;
            this.parent = parent;
            this.keySelector = keySelector;
            this.comparer = comparer != null ? comparer : Comparer<K>.Default;
            this.descending = descending;
        }

        internal override SequenceSorter<T> GetSequenceSorter(SequenceSorter<T> next) {
            SequenceSorter<T> sorter = new SequenceSorter<T, K>(keySelector, comparer,
descending, next);
            if (parent != null) sorter = parent.GetSequenceSorter(sorter);
            return sorter;
        }
    }

    internal abstract class SequenceSorter<T>
    {
        internal abstract void ComputeKeys(T[] elements, int count);

        internal abstract int CompareKeys(int index1, int index2);

        internal int[] Sort(T[] elements, int count) {
            ComputeKeys(elements, count);
            int[] map = new int[count];
            for (int i = 0; i < count; i++) map[i] = i;
```

```
            QuickSort(map, 0, count - 1);
            return map;
        }

        void QuickSort(int[] map, int left, int right) {
            do {
                int i = left;
                int j = right;
                int x = map[i + ((j - i) >> 1)];
                do {
                    while (CompareKeys(x, map[i]) > 0) i++;
                    while (CompareKeys(x, map[j]) < 0) j--;
                    if (i > j) break;
                    if (i < j) {
                        int temp = map[i];
                        map[i] = map[j];
                        map[j] = temp;
                    }
                    i++;
                    j--;
                } while (i <= j);
                if (j - left <= right - i) {
                    if (left < j) QuickSort(map, left, j);
                    left = i;
                }
                else {
                    if (i < right) QuickSort(map, i, right);
                    right = j;
                }
            } while (left < right);
        }
    }

    internal class SequenceSorter<T, K>: SequenceSorter<T>
    {
        internal Func<T, K> keySelector;
        internal IComparer<K> comparer;
        internal bool descending;
        internal SequenceSorter<T> next;
        internal K[] keys;

        internal SequenceSorter(Func<T, K> keySelector, IComparer<K> comparer, bool
descending, SequenceSorter<T> next) {
            this.keySelector = keySelector;
            this.comparer = comparer;
            this.descending = descending;
            this.next = next;
        }

        internal override void ComputeKeys(T[] elements, int count) {
            keys = new K[count];
            for (int i = 0; i < count; i++) keys[i] = keySelector(elements[i]);
            if (next != null) next.ComputeKeys(elements, count);
        }
```





59

```
        internal override int CompareKeys(int index1, int index2) {
            int c = comparer.Compare(keys[index1], keys[index2]);
            if (c == 0 && next != null) return next.CompareKeys(index1, index2);
            return descending ? -c : c;
        }
    }

    struct Buffer<T>
    {
        internal T[] items;
        internal int count;

        internal Buffer(IEnumerable<T> source) {
            T[] items = null;
            int count = 0;
            ICollection<T> collection = source as ICollection<T>;
            if (collection != null) {
                count = collection.Count;
                if (count > 0) {
                    items = new T[count];
                    collection.CopyTo(items, 0);
                }
            }
            else {
                foreach (T item in source) {
                    if (items == null) {
                        items = new T[4];
                    }
                    else if (items.Length == count) {
                        T[] newItems = new T[count * 2];
                        Array.Copy(items, 0, newItems, 0, count);
                        items = newItems;
                    }
                    items[count++] = item;
                }
            }
            this.items = items;
            this.count = count;
        }

        internal T[] ToArray() {
            if (count == 0) return new T[0];
            if (items.Length == count) return items;
            T[] result = new T[count];
            Array.Copy(items, 0, result, 0, count);
            return result;
        }
    }
}
```