|  |
| --- |
| , RD Dep. |
| Конспект и раздаточный материал  NET.C#.12 Введение в LINQ |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial version | Анжелика Кравчук |  |  |  |
| 1.1 | Review and corrections. | Владимир Тихон |  |  |  |

Contents

[1. Урок 1: Использование методов расширения и выражений запросов LINQ 3](#_Toc301449861)

[1.1. Введение в запросы LINQ 3](#_Toc301449862)

[1.2. Запросы к данным и построение набора результатов 5](#_Toc301449863)

[1.3. Фильтрация данных 7](#_Toc301449864)

[1.4. Упорядочивание данных 8](#_Toc301449865)

[1.5. Группировка данных и выполнение совокупных вычислений 9](#_Toc301449866)

[1.6. Объединение данных из различных наборов данных 11](#_Toc301449867)

[1.7. Выражения запросов C# 12](#_Toc301449868)

[1.8. Отложенное и раннее вычисление запросов 14](#_Toc301449869)

[1.9. Демонстрация: Методы расширения и выражения запросов LINQ 15](#_Toc301449870)

[2. Урок 2: Построение динамических запросов и выражений LINQ 16](#_Toc301449871)

[2.1. Динамический запрос LINQ 16](#_Toc301449872)

[2.2. Дерево выражений 17](#_Toc301449873)

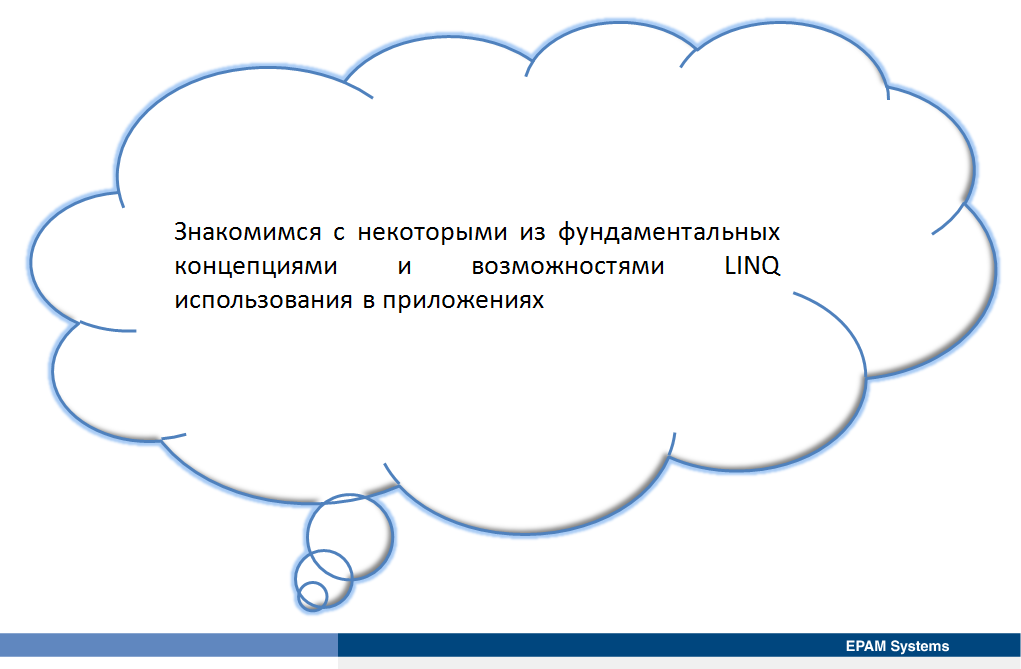
[2.3. Типы выражений 18](#_Toc301449874)

[2.4. Получение сведений о типах во время выполнения 21](#_Toc301449875)

[2.5. Компиляция и выполнение динамических запросов LINQ 22](#_Toc301449876)

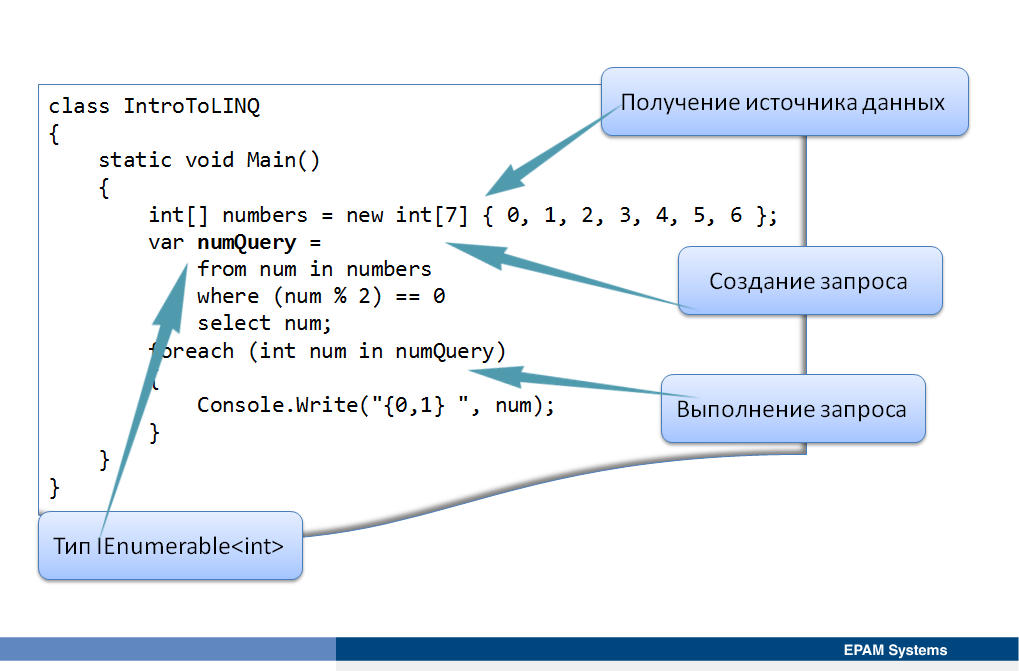
[2.6. Демонстрация: Построение динамических запросов и выражений LINQ 24](#_Toc301449877)

# Урок 1: Использование методов расширения и выражений запросов LINQ



Урок знакомит с возможностями LINQ .NET Framework. Используя LINQ, можно абстрагировать механизм, используемый приложением для запроса данных, от кода приложения. Урок знакомит с некоторыми из фундаментальных концепций и возможностей LINQ использования в приложениях.

## Введение в запросы LINQ



Большинство приложений выполняют какую-то обработку данных, которая может быть тривиальной. Например, получение списка параметров конфигурации приложения из файла. Однако, обработка данных может быть и сложной, например, если приложение выполняет массовое обновление до 5 миллионов записей в реляционной базе данных. Исторически сложилось так, что логика для выполнения таких операций была тесно связана с архитектурой приложения и структурой данных. Следовательно, если структура данных изменялась, то, как правило, необходимо было сделать значительные изменения в логике обработки данных.

Обработка различных данных, как правило, связана с построением запроса к ним. Запрос представляет собой выражение, получающее данные из источника данных. Запросы обычно выражаются на специальном языке запросов. К настоящему моменту разработаны различные языки для различных типов источников данных, например SQL для реляционных баз данных и XQuery для XML. Таким образом, разработчики вынуждены изучать новый язык запросов для каждого типа источника данных или формата данных, который они должны поддерживать. LINQ упрощает эту ситуацию, предлагая согласованную модель для работы с данными в различных видах источников данных и в различных форматах. В запросе LINQ работа всегда осуществляется с объектами. Для запросов и преобразований данных в XML-документах, базах данных SQL, наборах данных ADO.NET, коллекциях .NET и любых других форматах, для которых доступен поставщик LINQ, используются одинаковые базовые шаблоны кодирования.

Все запросы LINQ состоят из трех различных действий.

* Получение источника данных.
* Создание запроса.
* Выполнение запроса.

В следующем примере показано реализация этих трех частей операции запроса в исходном коде. В примере в качестве источника данных для удобства используется массив целых чисел; тем не менее, те же принципы применимы и к другим источникам данных.

class IntroToLINQ

{

static void Main()

{

// Три составляющих запроса LINQ:

// 1. Получение источника данных.

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

// 2. Создание запроса.

// переменная запроса numQuery является IEnumerable<int>

var numQuery =

from num in numbers

where (num % 2) == 0

select num;

// 3. Выполнение запроса.

foreach (int num in numQuery)

{

Console.Write("{0,1} ", num);

}

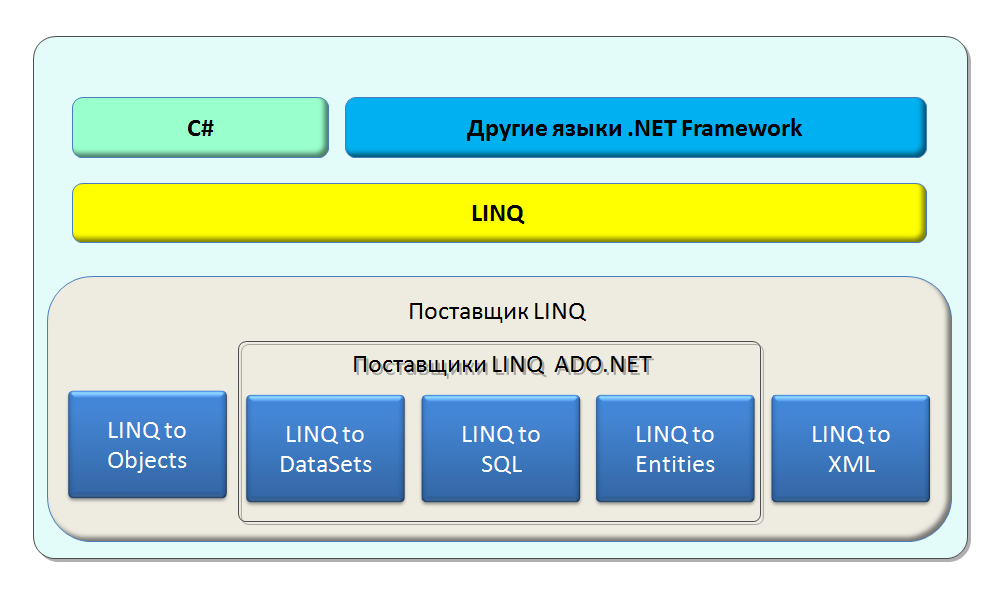
}

}

Запрос указывает, какую информацию нужно извлечь из источника или источников данных. При необходимости, запрос также указывает способ сортировки, группировки и формирования этих сведений перед возвращением. Запрос хранится в переменной запроса и инициализируется выражением запроса.

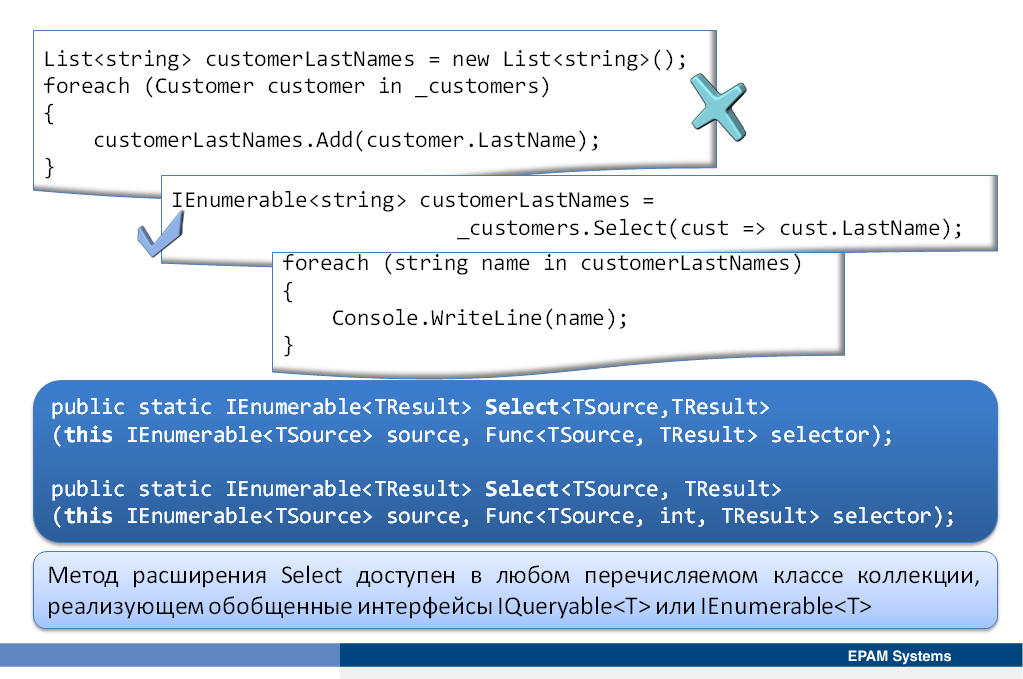
Использование LINQ обеспечивает высокоуровневое описание данных, которые должно получить приложение, но при этом точно не указыватся, как извлекаются данные. Поставщик LINQ принимает это описание и генерирует соответствующий код. Поставщик LINQ, по сути, является компонентом, реализующим необходимые интерфейсы, чтобы обеспечить коду возможности запросов. Каждый провайдер LINQ настроен для работы с определенным типом источника данных. В следующем списке приведены некоторые из доступных провайдеров LINQ:

* LINQ to XML
* LINQ to Objects
* LINQ to Entities
* LINQ to SQL
* LINQ to DataSets
* LINQ to SystemSearch



LINQ можно расширить путем создания собственных провайдеров.

## Запросы к данным и построение набора результатов



Технически LINQ to Objects – это набор классов, содержащих типичные методы обработки коллекций: поиск данных, сортировка, фильтрация и т.д, происходящие в памяти (in-memory data set). Ядром LINQ to Objects является статический класс Enumerable пространства имен System.Linq (Для использования System.Linq необходимо подключить сборку System.Core.dll). Этот класс содержит набор методов расширения интерфейса IEnumerable<T>.

namespace System.Linq

{

public static class Enumerable...

}

В этом разделе основное внимание уделяется тому, как использовать LINQ функциональность с помощью методов расширения, определеных в пространстве имен System.Linq. Альтернативный подход состоит в использовании выражений запросов LINQ.

Одной из самых основных функций LINQ является способность проектировать данные из коллекции. Например, для того, чтобы получить из списка объектов Customer второй список, содержащий только фамилию каждого клиента, можно использовать оператор foreach для перебора всей коллекции, извлекая необходимые данные, как показано в следующем примере кода.

IEnumerable<Customer> \_customers = new[]

{

new Customer{ FirstName = "Luka", LastName="Abrus", Age = 41},

new Customer{ FirstName = "Syed", LastName="Abbas", Age = 23},

new Customer{ FirstName = "Keith", LastName="Harris", Age = 59},

new Customer{ FirstName = "David", LastName="Pelton", Age = 25},

new Customer{ FirstName = "John", LastName="Peoples", Age = 37},

new Customer{ FirstName = "Toni", LastName="Poe", Age = 29},

new Customer{ FirstName = "Jeff", LastName="Price", Age = 74}

};

List<string> customerLastNames = new List<string>();

foreach (Customer customer in \_customers)

{

customerLastNames.Add(customer.LastName);

}

Гораздо более простым решением будет использование метода расширения Select, который обеспечивает LINQ.

Метод расширения Select доступен на любом перечисляемом классе коллекции, реализующем обобщенные интерфейсы IQueryable<T> или IEnumerable<T>. Простейшая форма метода Select принимает обобщенный делегат, который идентифицирует данные проекта. Самый простой способ для осуществления этого делегата – использовать лямбда-выражение.

public static IEnumerable<TResult> Select<TSource,TResult>(this IEnumerable<TSource>

source,Func<TSource, TResult> selector);

public static IEnumerable<TResult> Select<TSource, TResult>(this IEnumerable<TSource>

source,Func<TSource, int, TResult> selector);

В следующем примере показано использование метода расширения Select для получения списка фамилий из массива объектов Customer.

IEnumerable<string> customerLastNames = \_customers.Select(cust => cust.LastName);

Возвращаемое методом Select значение является ссылкой на перечислимую коллекцию, по которой можно итерироваться для выборки и обработки данных. В следующем примере перебирается коллекция, возвращенная вызовом предыдущего метода Select, а затем отображаются результаты.

foreach (string name in customerLastNames)

{

Console.WriteLine(name);

}

Если необходимо вернуть данные из более чем одного свойства, можно создать анонимный тип. Анонимный тип это тип без названия, которое компилятор генерирует автоматически. Компилятор неявно создает тип, не требующий явного объявления, которое ожидается от стандартного типа, такого как класс. для создания экземпляра анонимного типа необходимо использовать ключевое слово new, за которым следует пара фигурных скобок, в которых определяются имена полей и значения, которые будет содержать тип. Типы полей компилятор выводит из значений, предоставленых для них.

В следующем примере показано использование метода расширения Select для извлечения свойств FirstName и LastName в анонимный тип и возвращения перечислимой коллекции этого типа. Затем выполняется итерация по коллекции для отображения результатов.

var customerNames = \_customers.Select(cust =>

new { FirsName = cust.FirstName, LastName = cust.LastName });

foreach (var customer in customerNames)

{

Console.WriteLine("{0} {1}", customer.FirsName, customer.LastName);

}

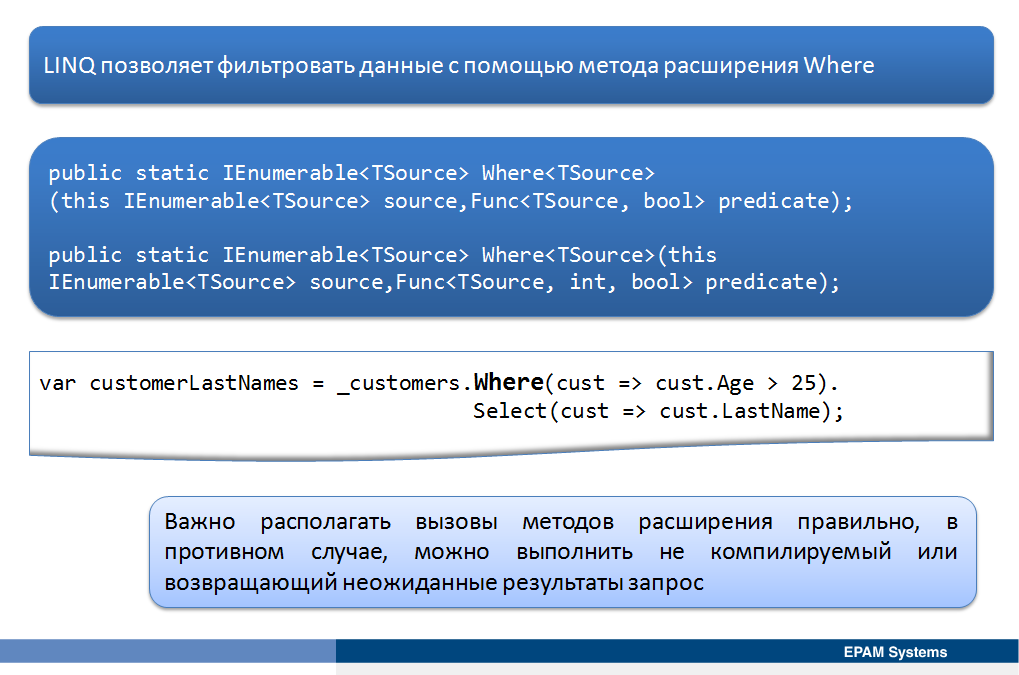
Именами полей в анонимном типе может быть любой допустимый идентификатор, они не обязаны быть такими же, как имена полей в базовом типе.

В этом примере заранее не известно название типа, который возвращает метод Select, таким образом переменные customerNamesAnonim и customer определены как var.

http://go.microsoft.com/fwlink/?LinkId=192978

http://go.microsoft.com/fwlink/?LinkId=192979

## Фильтрация данных



Метод расширения Select позволяет указать поля, которые нужно вернуть из перечислимой коллекции. Однако можно ограничить элементы, возвращаемые из коллекции. LINQ позволяет фильтровать данные с помощью метода расширения Where.

Синтаксически метод расширения Where аналогичен методу Select, метод ожидает делегат и возвращает объекта IEnumerable. Делегат должен принимать экземпляр данных, которые вычисляются, и должен возвращать логическое значение, чтобы указать, являются ли эти данные включенными в перечислимое результирующее множество.

public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,

Func<TSource, bool> predicate);

public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,

Func<TSource, int, bool> predicate);

Как и в методе Select, самый простой способ обеспечить этот делегат это использовать лямбда-выражение. Расширяющий метод Where возвращает перечислимый результат, к котороу можно применить метод расширения Select и применить проекцию. В следующем примере показано использование метода расширения Where для возвращения коллекции фамилий клиентов, которым более 25 лет.

var customerLastNames = \_customers.Where(cust => cust.Age > 25).

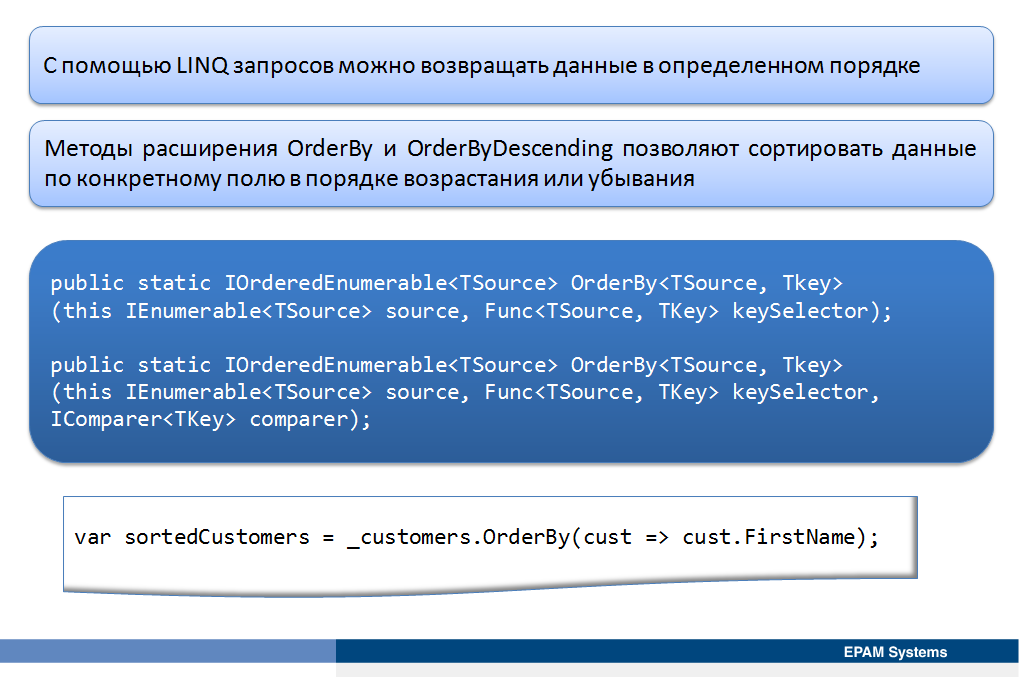
Select(cust => cust.LastName);

В предыдущем примере, методы расширения применяется следующим образом.

* Метод расширения Where применяется к массиву customers для выполнения фильтрации; он возвращает объект IEnumerable, содержащий только записи, соответствующие условию. Возвращаемые записи включают все поля объекта Customer.
* Затем применяется метод расширения Select, указывающий, что должно быть возвращено только свойство LastName.

Важно располагать вызовы методов расширения правильно, в противном случае, можно выполнить не компилируемый или возвращающий неожиданные результаты запрос. В предыдущем примере, если выполнить попытку вызвать метод расширения Where после метода расширения Select, запрос не будет компилироваться. Это объясняется тем, что лямбда-выражение в методе расширения Where относится к полю Age объекта Customer, а метод расширения Select проецирует только поле LastName. Если метод расширения Select проецирует весь объект Customer, запрос будет компилировать и запускаться как обычно.

## Упорядочивание данных



С помощью LINQ запросов можно фильтровать строки данных и выбирать конкретные поля из типов, можно также возвращать данные в определенном порядке. LINQ предоставляет схожую с оператором OrderBySQL функциональность с помощью методов расширения OrderBy, OrderByDescending, ThenBy и ThenByDescending.

Большинство методов расширения, таких как Select, Where и OrderBy являются обобщенными методами. Это означает, что компилятор работает, определяя типы для использования на основе контекста, но когда существует потенциальная двусмысленность, может понадобиться указать соответствующие параметры типа.

Методы расширения OrderBy и OrderByDescending позволяют сортировать данные по конкретному полю в порядке возрастания или убывания. Как и в других методах расширения, эти два метода ожидают делегат, который определяет поле или выражение для сортировки данных.

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>

(this IEnumerable<TSource> source,

Func<TSource, TKey> keySelector);

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>

(this IEnumerable<TSource> source,

Func<TSource, TKey> keySelector,

IComparer<TKey> comparer);

public static IOrderedEnumerable<TSource> OrderByDescending<TSource, TKey>

(this IEnumerable<TSource> source,

Func<TSource, TKey> keySelector);

public static IOrderedEnumerable<TSource> OrderByDescending<TSource, TKey>

(this IEnumerable<TSource> source,

Func<TSource, TKey> keySelector,

IComparer<TKey> comparer);

В следующем примере кода показано, как использовать метод расширения OrderBy для сортировки массива объектов Customer по полю FirstName в порядке возрастания.

var sortedCustomers = \_customers.OrderBy(cust => cust.FirstName);

В следующем примере показано использование метода расширения OrderByDescending для сортировки массива объектов по полю FirstName в порядке убывания.

var sortedCustomers = \_customers.OrderByDescending(cust => cust.FirstName);

Хотя методы расширения OrderBy и OrderByDescending позволяют выполнять основные сортировки, иногда можно выполнить дополнительную сортировку в том же операторе. Методы расширения OrderBy и OrderByDescending возвращают объект IOrderedEnumerable, который предоставляет два дополнительных метода расширения: ThenBy и ThenByDescending. Методы расширения ThenBy и ThenByDescending позволяют указать дополнительные ключи сортировки для данных.

public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>

(this IOrderedEnumerable<TSource> source,

Func<TSource, TKey> keySelector);

public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>

(this IOrderedEnumerable<TSource> source,

Func<TSource, TKey> keySelector,

IComparer<TKey> comparer);

public static IOrderedEnumerable<TSource> ThenByDescending<TSource, TKey>

(this IOrderedEnumerable<TSource> source,

Func<TSource, TKey> keySelector);

public static IOrderedEnumerable<TSource> ThenByDescending<TSource, TKey>

(this IOrderedEnumerable<TSource> source,

Func<TSource, TKey> keySelector,

IComparer<TKey> comparer);

В следующем примере кода показано использование методов расширения OrderBy и OrderByDescending. В примере используется метод OrderBy для сортировки результатов по полю FirstName, а затем используется метод ThenBy для сортировки записей по полю Age.

var sortedCustomers = \_customers.OrderBy(cust => cust.FirstName).

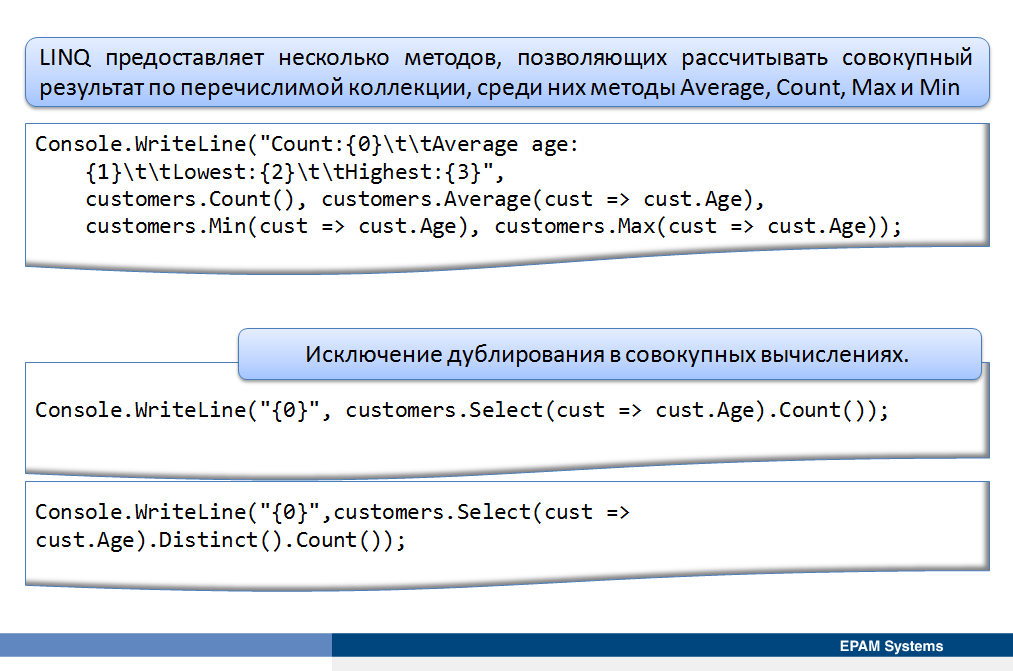
ThenBy(cust => cust.Age);

Метод расширения ThenByDescending позволяет применять дополнительные сортировки последовательности по убыванию.

var sortedCustomers = \_customers.OrderByDescending(cust => cust.FirstName).

ThenByDescending(cust => cust.Age);

## Группировка данных и выполнение совокупных вычислений



LINQ предоставляет несколько методов, позволяющих рассчитывать совокупный результат по перечислимой коллекции, среди них методы Average, Count, Max и Min. Методы Average, Count, Max и Min. принимают делегат, указывающий поле, по которому вычисляется совокупное значение. Метод Count принимает делегат, определяющий предикат для вычисления и включает элемент в вычисление только, если предикат возвращает значение true. Как правило метод Count используют для определения общего количество строк в коллекции, и поэтому делегат часто опускается (все совокупные методы перегружены).

В следующем примере показано использование каждого из указанных методов с массивом объектов customer. Пример отображает общее число клиентов, их средний возраст, а также минимальный и максимальный возраст.

IEnumerable<Customer> customers = new[]

{

new Customer{ FirstName = "Luka", LastName="Abrus", Age = 41},

new Customer{ FirstName = "Syed", LastName="Abbas", Age = 23},

new Customer{ FirstName = "Keith", LastName="Harris", Age = 59},

new Customer{ FirstName = "David", LastName="Pelton", Age = 29},

new Customer{ FirstName = "John", LastName="Peoples", Age = 37},

new Customer{ FirstName = "Toni", LastName="Poe", Age = 29},

new Customer{ FirstName = "Jeff", LastName="Price", Age = 74}

};

Console.WriteLine("Count:{0}\t\tAverage age:{1}\t\tLowest:{2}\t\tHighest:{3}",

customers.Count(), customers.Average(cust => cust.Age),

customers.Min(cust => cust.Age), customers.Max(cust => cust.Age));

Совокупные вычисления можно выполнить непосредственно на фоне перечислимой коллекции, но часто требуется вычислить совокупное значение для разных групп данных; например, определить, сколько клиентов внутри определенных возрастных диапазонов. Можно разделить перечислимую коллекцию на группы, используя метод расширения GroupBy. Метод GroupBy ожидает делегат, показывающий, как группируются данные; делегированный метод возвращает селектор, и все элементы, имеющие значения, совпадающие со значением селектора, помещаются в одну группу. После чего можно выполнить совокупные методы по каждой группе.

Значение, возвращаемое методом GroupBy является перечислимой коллекцией объектов, реализующей интерфейс IGrouping. Интерфейс IGrouping представляет коллекцию элементов, имеющих общее значение ключа, и предоставляют доступ к этим ключам, используя свойство Key.

В следующем примере используется метод GroupBy, разделяющий клиентов на группы, определяемые их возрастом. Затем пример отображает общее число клиентов в каждой группе и селектор, определяющий группу. Следует отметить, что значение селектора доступно через свойство группы Key. Пример назначает группы с помощью этого ключа, а также отображает ключ как часть вывода.

var customersGroupedByAgeRange = customers.GroupBy(cust =>

{

if (cust.Age < 20)

return "age < 20";

if (cust.Age >= 20 && cust.Age < 40)

return "age >= 20 and < 40";

if (cust.Age >= 40 && cust.Age < 60)

return "age >= 40 and < 60";

if (cust.Age >= 60)

return "age >= 60";

return "Error";

});

foreach (var cust in customersGroupedByAgeRange.OrderBy(cust => cust.Key))

{

Console.WriteLine("{0}\t\t{1}", cust.Key, cust.Count());

}

Код генерирует следующий результат.

age >= 20 and < 40      4

age >= 40 and < 60      2

age >= 60      1

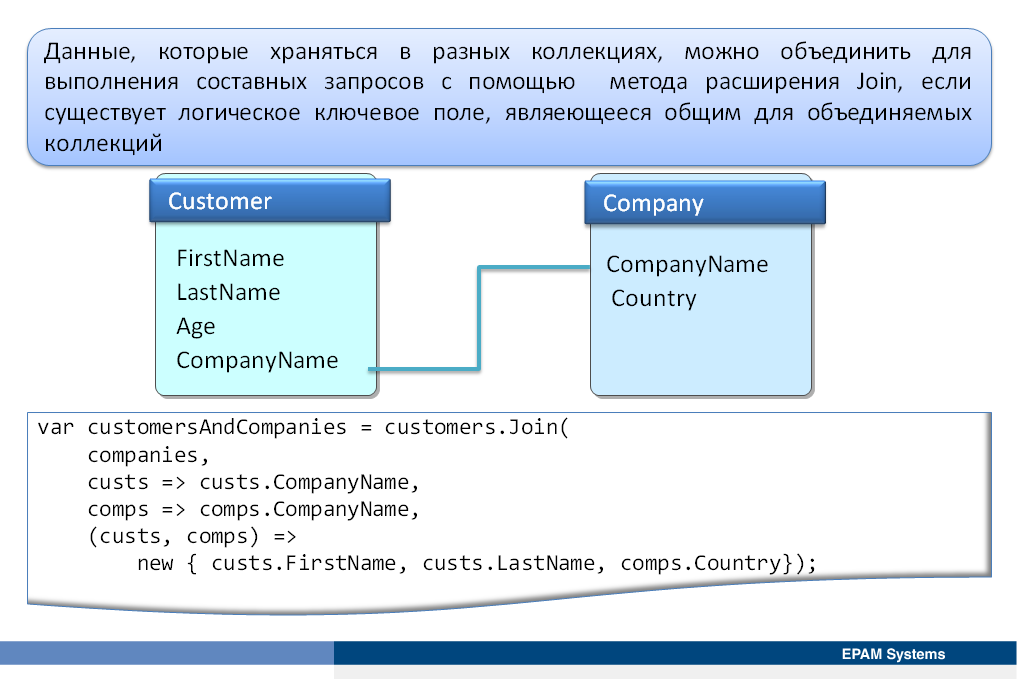
Совокупный методы, такие как Count или Average включает в себя все соответствующие элементы в своих вычислениях. Однако иногда нужно отбросить любые повторяющиеся значения из этих вычислениий. Запрос в следующем примере подсчитывается число возрастов, которые находятся в массиве customers.

Console.WriteLine("{0}", customers.Select(cust => cust.Age).Count());

Если два или более клиентов имеют однин и тот же возраст, этот возраст считается дважды. Исключить повторяющиеся значения можно с помощью метода расширения Distinct перед выполнением агрегации.

Console.WriteLine("{0}",customers.Select(cust => cust.Age).Distinct().Count());

## Объединение данных из различных наборов данных



Данные, которые храняться в разных коллекциях, можно объединить для выполнения составных запросов, если существует логическое ключевое поле, являеющееся общим для обеих коллекций. Для этой цели LINQ предоставляет метод расширения Join.

В следующем примере объявляются два массива. Первый содержит информацию о клиентах, включая фамилию, имя, возраст и имя компании, в которой они работают. Второй массив содержит информацию о компаниях, включа название и страну, где находится компания.

IEnumerable<Customer> customers = new[]

{

new Customer{ FirstName = "Luka", LastName="Abrus", Age = 41, CompanyName = "Contoso"},

new Customer{ FirstName = "Syed", LastName="Abbas", Age = 23, CompanyName = "Fabrikam"},

new Customer{ FirstName = "Keith", LastName="Harris", Age = 59, CompanyName = "Contoso"},

new Customer{ FirstName = "David", LastName="Pelton", Age = 41, CompanyName = "Contoso"},

new Customer{ FirstName = "John", LastName="Peoples", Age = 23, CompanyName = "Contoso"},

new Customer{ FirstName = "Toni", LastName="Poe", Age = 29, CompanyName = "Fabrikam"},

new Customer{ FirstName = "Jeff", LastName="Price", Age = 23, CompanyName = "Fabrikam"}

};

IEnumerable<Company> companies = new[]

{

new Company{CompanyName = "Contoso", Country = "United Kingdom"},

new Company{CompanyName = "Fabrikam", Country = "United States"}

};

С помощью метода расширения Join можно получить перечислимую коллекцию, содержащую данные из обоих массивов customers и companies, объединяя их посредством поля CompanyName.

Чтобы использовать метод расширения Join, предоставляютя следующие параметры:

* Перечислимая коллекция, с которой происходит объединение.
* Метод, определяющий общие поля, идентифицирующие метод расширения Select.
* Метод, определяющий общие ключевые поля, на основании которых объединяются различные наборы данных.
* Метод, определяющий поля, требуемые от результирующего перечислимого набора, возвращающемого методом расширения Join.

В следующем примере показано использование метода расширения Join для объединения массивов customers и companies посредством поля CompanyName и отображения фамилии и имени клиента, а также страны, в которой он проживает.

var customersAndCompanies = customers.Join(companies,

custs => custs.CompanyName,

comps => comps.CompanyName,

(custs, comps) => new { custs.FirstName, custs.LastName, comps.Country });

foreach (var item in customersAndCompanies)

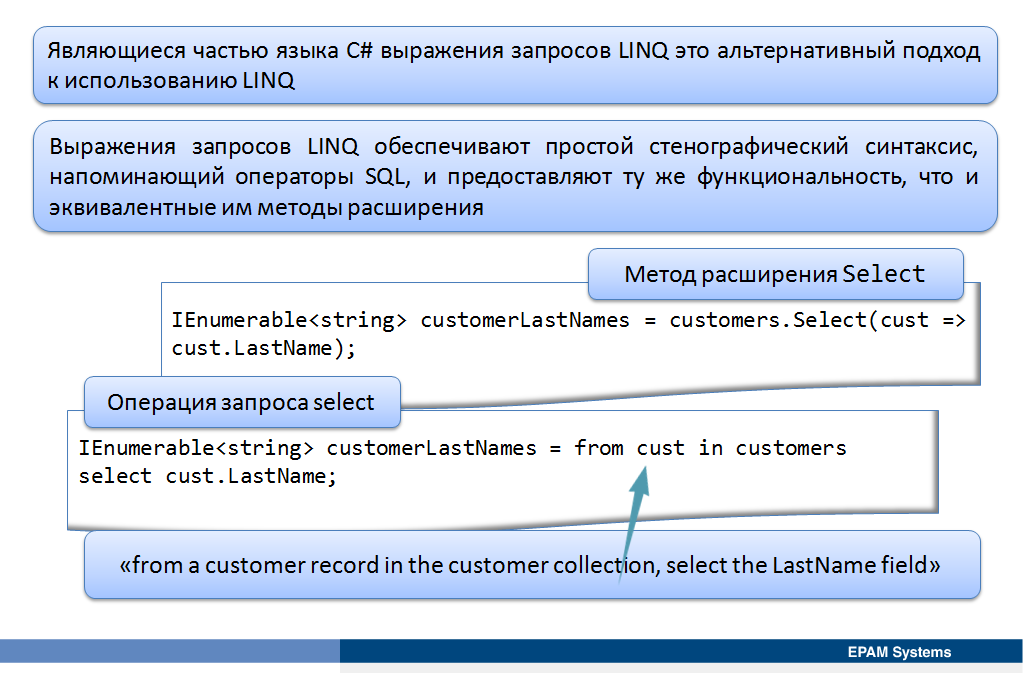
{

Console.WriteLine(item);

}

Метод Join возвращает коллекцию IQueryable, которую можно использовать в сочетании с другими методами расширения LINQ, такими как Select, Where и OrderBy.

## Выражения запросов C#



Методы расширения LINQ и типы, определенные в пространстве имен System.Linq, являются очень мощным динамическим средством, однако синтаксис их может быть довольно громоздким, что может легко добавлять в код ошибки, которые трудно обнаружить и исправить.

Разработчики C# признали этот факт и предоставили альтернативный подход к использованию LINQ через выражения запросов LINQ, являющиеся частью языка C#. Выражения запросов LINQ обеспечивают простой стенографический синтаксис, напоминающий операторы SQL; и предоставляют ту же функциональность, что и эквивалентные им методы расширения. Во время компиляции, сокращенный синтаксис выражений запросов компилируется в соответствующий код, использующий методы расширения LINQ.

Например, для выборки списка фамилий из массива объектов Customer можно использовать метод расширения.

IEnumerable<string> customerLastNames = customers.Select(cust => cust.LastName);

Однако, можно использовать выражения запроса from, in и select для достижения того же результата, как показано в следующем примере.

IEnumerable<string> customerLastNames = from cust in customers select cust.LastName;

Подход выражений запросов значительно более интуитивен и прост. Предыдущую операцию можно прочитать следующим обазом «from a customer record in the customer collection, select the LastName field».

Выражение запроса должна подчиняться следующим правилам:

1. Выражение запроса должно начинаться с конструкции from, которая указывает на обрабатываемую коллекцию.
2. Затем выражение запроса может содержать ноль или более конструкции from, let или where. Конструкция let представляет переменную и присваивает ей значение. Конструкция where фильтрует элементы коллекции.
3. Затем выражение запроса может включать ноль или более конструкций orderby, с полями сортировки и необязательным указанием на направление упорядочивания. Направление может быть ascending или descending.
4. Затем следует конструкция select или group.
5. Наконец, в оставшейся части выражения запроса может следовать необязательная конструкция продолжения. Такой конструкцией является into.

В сравнении с SQL, порядок предложения в запросе LINQ отличается; оператор from всегда предшествует оператору select.

Как и в методах расширения в выражениях запроса можно воспользоваться анонимными типами и выводить новые объекты для хранения подмножества полей из оригинального набора данных. В следующем примере показано использование выражения запроса и анонимных типов.

var custs = from cust in customers select new { cust.FirstName, cust.LastName };

Почти для каждого метода расширения LINQ существует эквивалентное выражение запроса. В следующем списке сравнивается синтаксис методов расширения Where, OrderBy, GroupBy и Join и эквивалентных выражений запроса:

* Следующие примеры иллюстрируют метод расширения Where и соответствующее выражение запроса where.

//Where extension method example

var customersOver25 = customers.Where(cust => cust.Age > 25);

//where query operator example

var customersOver25 = from cust in customers where cust.Age > 25 select cust;

* Следующие примеры иллюстрируют метод расширения OrderBy и соответствующее выражение запроса orderby.

//OrderBy extension method example

var sortedCustomers = customers.OrderBy(cust => cust.FirstName);

//orderby query operator example

var sortedCustomers = from cust in customers orderby cust.FirstName select cust;

* Следующие примеры иллюстрируют метод расширения GroupBy и соответствующее выражение запроса groupby.

//GroupBy extension method example

var customersGroupedByAge = customers.GroupBy(cust => cust.Age);

//groupby query operator example

var customersGroupedByAge = from cust in customers group cust by cust.Age;

* Следующие примеры кода иллюстрируют метод расширения Join и соответствующее выражение запроса join

//Join extension method example

var customersAndCountries = customers.Join( companies,

cust => cust.CompanyName,

comp => comp.CompanyName,

cust, comp) =>

new {cust.FirstName,cust.LastName,comp.Country});

//join query operator example

var customersAndCountries1 = from cust in customers

join comp in companies on cust.CompanyName

equals comp.CompanyName

select new {cust.FirstName, cust.LastName, comp.Country};

Совокупные вычисления можно выполнять на результатах, которые возвращают LINQ запросы, использующие выражения запросов. Запрос, использующий выражение запроса LINQ, возвращает перечислимое результирующее множество. Для использования совокупных методов, таких, как Count, Max, Min и Distinct, необходимо обернуть запрос LINQ в скобки а затем применить соответствующий метод расширения. В следующем примере показано использование метода расширения Count с простым запросом, использующим выражение запроса.

var customerCount = (from cust in customers select cust).Count();

Такой же подход использовать с другими краткими методами расширения. Ниже приведены примеры использования методов расширения Max, Min и Distinct с выражениями запросов.

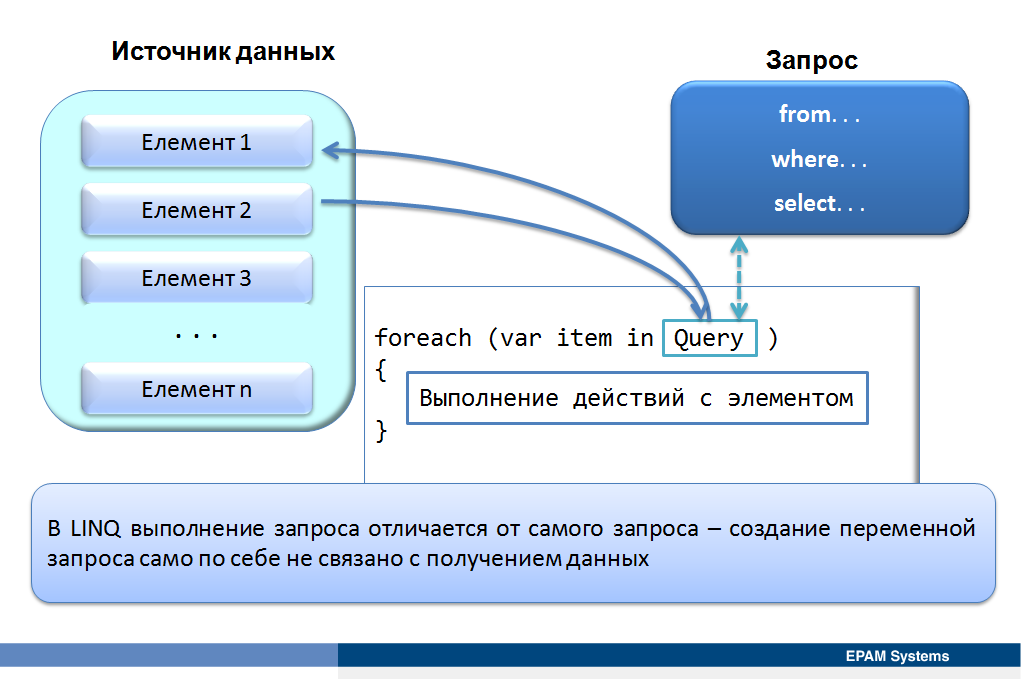
var maxAge = (from cust in customers select cust.Age).Max();

var minAge = (from cust in customers select cust.Age).Min();

var possibleAges = (from cust in customers select cust.Age).Distinct();

<http://go.microsoft.com/fwlink/?LinkId=192980>

## Отложенное и раннее вычисление запросов



При использовании LINQ определить перечислимую коллекцию можно либо с помощью методов расширения LINQ или с помощью операций запросов. Во время выполнения методов расширения LINQ и операций запросов приложение не строит коллекцию; данные получаются только тогда, когда происходит итерирование по коллекции. Это означает, что данные в оригинальной коллекции могут измениться после выполнения запросов LINQ, но до извлечения данных, определяемых запрос; поэтому данные всегда актуальны. Запрос в следующем примере определяет перечислимую коллекцию американских компаний.

var usCompanies = from a in companies where String.Equals(a.Country, "United States") select a.CompanyName;

Данные в массиве companies не будут найдены, а какие-либо условия, указанные в операторе where, не вычислены до тех пор, пока не будет перебирана коллекция usCompanies.

foreach (string name in usCompanies)

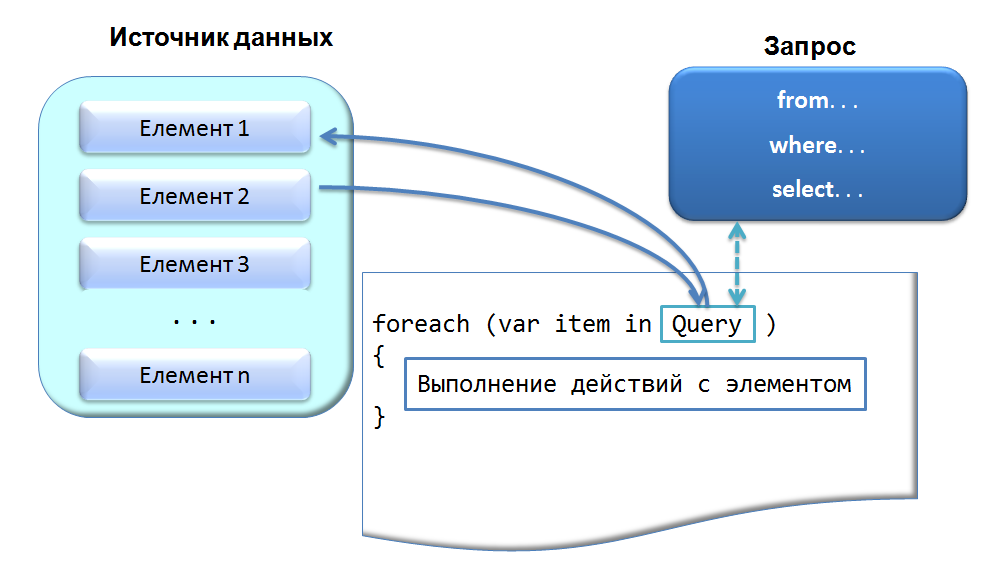
{

Console.WriteLine(name);

}

При изменении данных в массиве companies после определения коллекции usCompanies, но, прежде чем выполнить итерацию по коллекции (например, при добавлении новой компании), данные будут обновлены. Эта стратегия называется отложенным вычислением.

В LINQ выполнение запроса отличается от самого запроса; другими словами, создание переменной запроса само по себе не связано с получением данных. На рисунке показана завершенная операция запроса.



Можно форсировать вычисление запросов LINQ и генерировать статическую, кэшированную коллекцию. Эта коллекция будет копией исходных данных и не изменится при изменении данных в наборе. LINQ предоставляет метод ToList для создания статического объекта List, содержащего кэшированную копию данных.

var usCompanies = from a in companies.ToList()

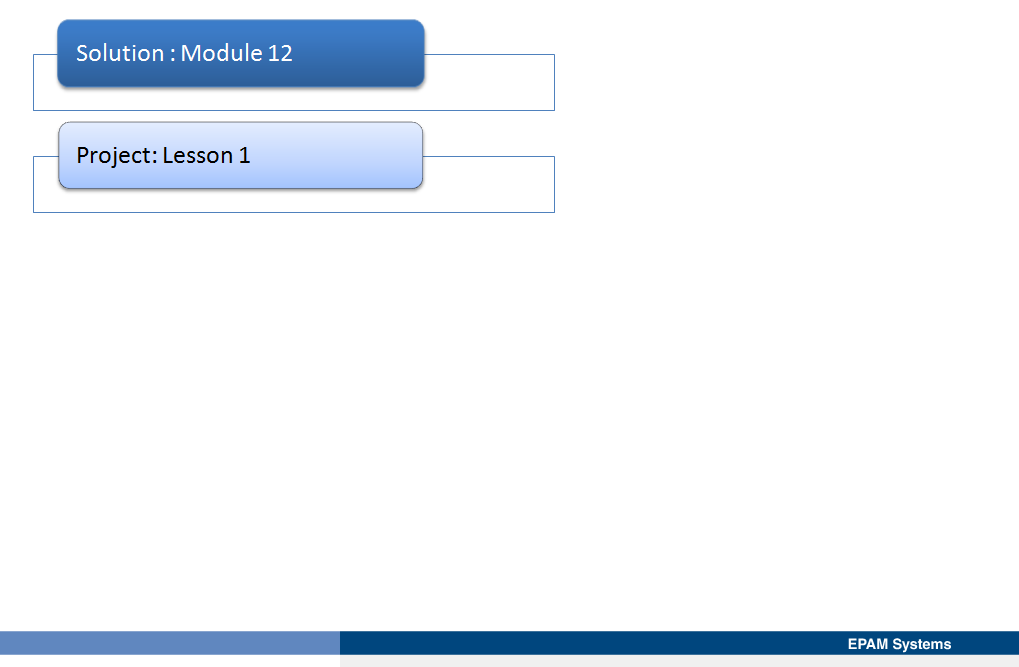
where String.Equals(a.Country, "United States")

select a.CompanyName;

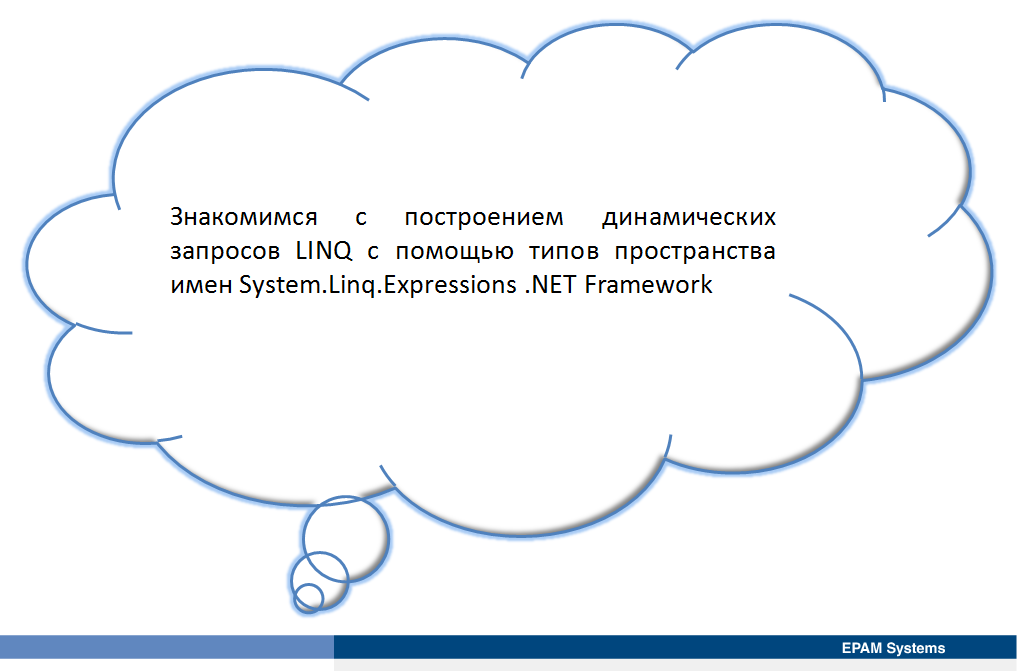
На этот раз, список компаний, является фиксированным при определении запроса. Если добавить в массив companies больше американских компаний, они не появятся при переборе коллекции usCompanies. LINQ также предоставляет метод ToArray, который сохраняет в кэше коллекцию в виде массива.

<http://go.microsoft.com/fwlink/?LinkId=192981>

## Демонстрация: Методы расширения и выражения запросов LINQ

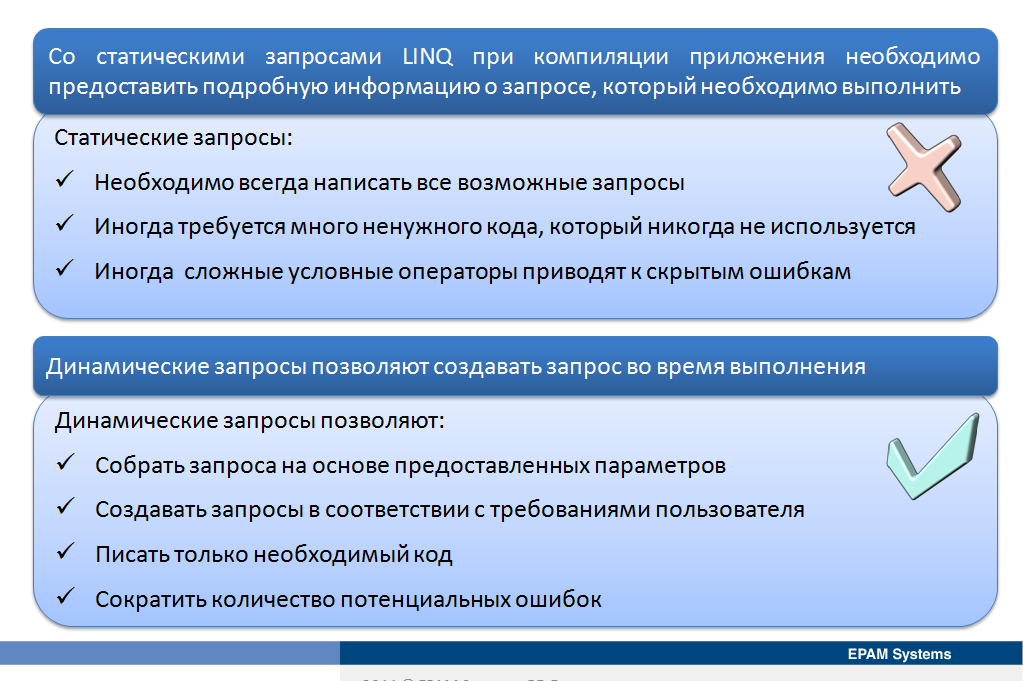


# Урок 2: Построение динамических запросов и выражений LINQ



При использовании LINQ можно запрашивать данные из кода приложения, не беспокоясь о реализации базового источника данных. Однако со статическими запросами LINQ необходимо предоставить подробную информацию о запросе, который необходимо выполнить при компиляции приложения. Запросы LINQ можно строить динамически. Такой подход позволяет оптимизировать запросы путем формирования их точных требований, которые указывает пользователь приложения. Урок описывает, как строить динамические запросы LINQ с помощью типов пространства имен System.Linq.Expressions .NET Framework.

## Динамический запрос LINQ



LINQ предоставляет очень мощный механизм, позволяющий отделить бизнес-логику приложения от логики, необходимой для получения данных, которые использует приложение. В большинстве ситуаций можно использовать статические запросы LINQ, но могут быть и случаи, когда критерии запроса, порядок, в котором требуется данные, или даже получаемые данные не известны до времени выполнения. Например, форма запроса может зависеть от ввода пользователя или результатов некоторых других видов обработки.

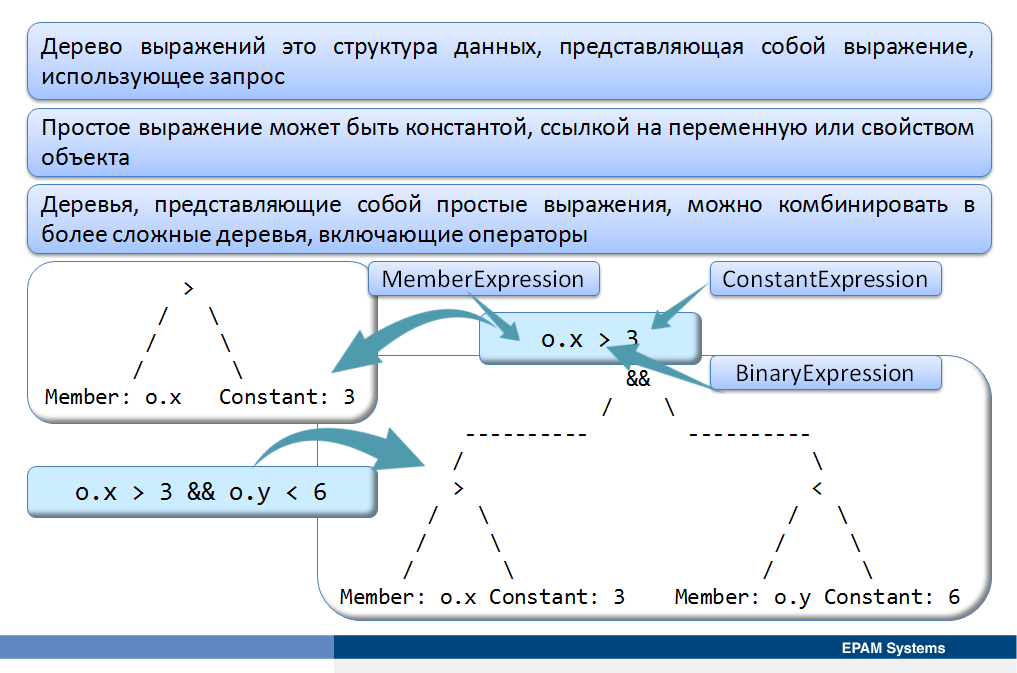
Например, может потребоваться запросить таблицы базы данных, в которой хранятся результаты тестов, выполненных студентами. Эта таблица может содержать поля, хранящие имя студента, название теста и оценку теста для этого студента. С помощью этой простой схемы, можно разрабатывать приложения для запроса базы данных и применять фильтры к данным. Простые запросы можно выполнять с помощью статического LINQ.

Однако предположим, что нужно сделать приложение более гибкими. Например, если нужно отфильтровать результаты для студентов, получивших оценки в определенном диапазоне для одного или нескольких тестов и отсортировать их по совокупности ключей (по возрастанию/убыванию названия теста, оценке, имени студента или любому другому сочетанию).

В некоторых случаях можно получить все три поля из базы данных, но в других случаях, может существовать заинтересованность только в подмножестве полей. Один из вариантов заключается в написании нескольких запросов, а затем использовании операторов if или switch для выбора запроса для использования, однако такой подход влечет за собой написание множества кода, что увеличивает риск появления ошибок. Другой вариант состоит в извлечении всех данных коллекции в память, а затем программном получении соответствующих данных из этой коллекции, но такой подход также потенциально подвержен ошибкам и не использует всю мощь LINQ.

Использование динамических запросов LINQ помогает решить эту проблему. При разработке динамических запросов LINQ представление запроса строится в виде дерева выражений (expression tree), которое компилируется во время выполнения, а затем выполняется. Тогда запросы точно представляют потребности пользователя без необходимости разработки нескольких запросов или обработки данных вручную в памяти.

## Дерево выражений



Дерево выражений это структура данных, представляющая собой выражение, использующее запрос. Простое выражение может быть константой, ссылкой на переменную или свойством объекта. Можно комбинировать деревья, представляющие собой простые выражения, в более сложные деревья, включающие операторы.

Например, выражение o.x > 3 состоит из двух меньших выражений (переменной-члена x объекта o и постоянной 3), объединенных в большее выражение с помощью операции > (больше чем). Следующая диаграмма показывает, каким образом можно себе представить выражение.

           >

         /   \

        /     \

       /       \

Member: o.x   Constant: 3

Соответствующее дерево выражений для этого выражения состояит из объекта MemberExpression, который ссылается на поле x объекта o, и объекта ConstantExpression, представляющего постоянное значение 3; эти объекты объединены в объект BinaryExpression, сравнивающего их с помощью операции > (больше чем). Дерево выражений может представлять постоянную, член или комбинировать деревья выражений вместе с помощью операций. Так, например, дерево выражения в следующей диаграмме представляет выражение o.x > 3 && o.y < 6

                       &&

                     /    \

          ----------        ----------

         /                            \

         >                            <

       /   \                        /   \

      /     \                      /     \

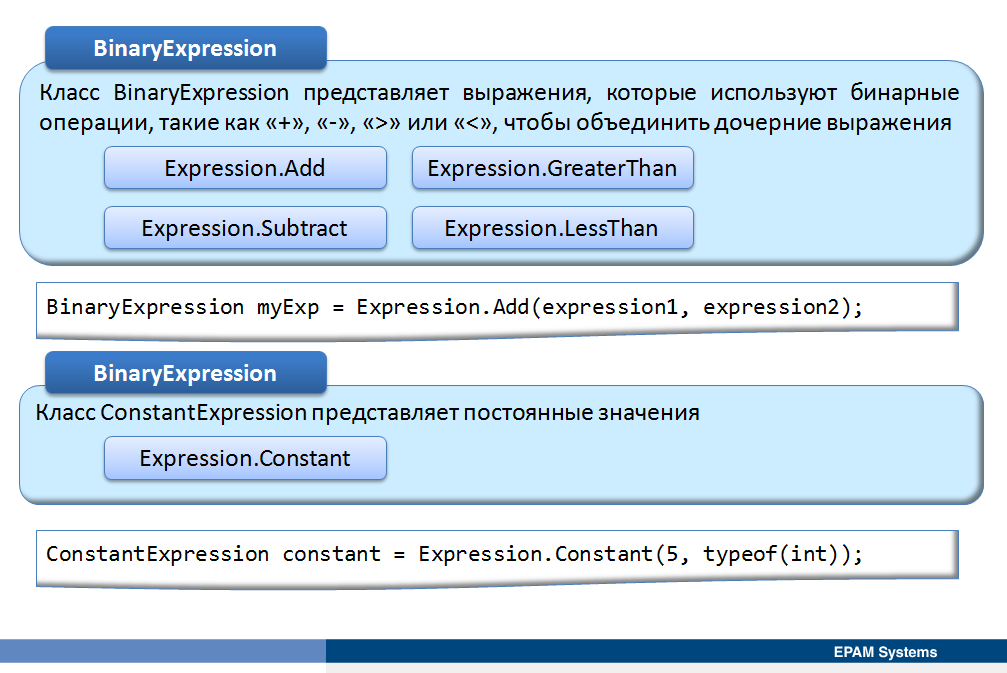
     /       \                    /       \

Member: o.x Constant: 3    Member: o.y Constant: 6

Деревья выражений обеспечивают очень гибкий механизм для разработки выражений, которые можно использовать как часть запроса LINQ. Можно строить деревья выражений, представляющие лямбда-выражения, на которые затем можно сослаться в запросе LINQ. Можно скомпилировать дерево выражения, представляющее лямбда-выражение во время выполнения, а затем вызывать его так же, как обычное лямбда-выражение.

 http://go.microsoft.com/fwlink/?LinkId=192982

## Типы выражений



Каждый узел в дереве выражений является объектом выражения (expression object). Пространство имен System.Linq.Expressions в библиотеке классов .NET Framework определяет типы выражений, которые можно использовать для представления любого допустимого выражения C#. Все эти типы наследуются от класса Expression пространства имен System.Linq.Expressions.

При разработке WPF приложений следует знать, что WPF определяет тип Expression. Поэтому, если необходимо использовать тип Expression из пространства имен System.Linq.Expressions, полезно создать псевдоним для пространства имен и использовать псевдоним, чтобы избежать двусмысленности.

В следующем примере кода показано, как добавить псевдоним для пространства имен System.Linq.Expressions имени Expressions.

using Expressions = System.Linq.Expressions;

При построении простых выражений используется соответствующий тип выражения. Простые выражения можно комбинировать в более сложные, используя класс Expressions. Класс Expressions это абстрактный тип, действующий как родитель всех типов выражения. Класс Expressions обеспечивает большое количество статических фабрик-методов для объединения выражений вместе в новые деревья выражений с помощью любых операций, доступных в языке C#. Например, метод-фабрика Expression.Add[[1]](#footnote-1) принимает два объекта Expression и, объединяя их с помощью оператора сложения (+), создает новое дерево выражений.

**Класс BinaryExpression.** Класс BinaryExpression представляет выражения, которые используют бинарные операции, такие как «+», «-», «>» или «<», чтобы объединить дочерние выражения. Такие методы, как Expression.Add, Expression.Subtract, Expression.GreaterThan и Expression.LessThan возвращают объект BinaryExpression. В следующем примере показано создание объекта BinaryExpression, складывающего два значения.

// Assume that expression1 and expression2 are expression tree objects.

BinaryExpression myExp = Expression.Add(expression1, expression2);

**Класс ConstantExpression.** Класс ConstantExpression представляет постоянные значения. Для создания объекта ConstantExpression используется статический метод Expression.Constant. При этом необходимо предоставить значение и, возможно, тип System.Type объекта в качестве аргументов для метода Constant. Следует обратить внимание, что можно использовать оператор typeof для возвращения типа. В следующем примере показано определение выражения ConstantExpression для целого значения 5.

ConstantExpression constant = Expression.Constant(5, typeof(int));

**Класс MemberExpression.** Класс MemberExpression позволяет обращаться к свойствам или полям объекта. Например, можно использовать объект MemberExpression для представления выражения x.member в бинарном выражении x.member > 20.

Для создания экземпляра класса MemberExpression используется статический метод MakeMemberAccess класса Expression. Метод MakeMemberAccess принимает два аргумента: ссылку на объект, который содержит член в форме объекта Expression и ссылку на себя в форме объекта System.Reflection.MemberInfo. Объект Expression, представляющий объект, может быть ссылкой на параметр, передаваемый в лямбда-выражение. или Constant выражением, если нужно сослаться на объект, который был создан за пределами лямбда-выражения.

Можно использовать отражение для создания объекта MemberInfo, представляющего член объекта.

В следующем примере кода показано создание объекта MemberExpression для доступа к свойству объекта myData типа MyType.

// Assume that the propertyInfo object is a valid instance of the

// MemberInfo class that references a field that the MyType type exposes.

MyType myData = ...;

MemberExpression member = Expression.MakeMemberAccess(Expression.Constant(myData),

propertyInfo);

Иногда может потребоваться доступ к статическому члену типа. В этом случае можно указать ссылку null на конкретный экземпляр объекта в качестве первого параметра метода MakeMemberAccess. В следующем примере кода показано создание объекта MemberExpression для доступа к статическому свойству.

// Assume that the propertyInfo object is a valid MemberInfo object

// that references a static property of a type.

MemberExpression staticProperty = Expression.MakeMemberAccess(null, propertyInfo);

**Класс UnaryExpression.** Класс UnaryExpression представляет выражения, основаные на унарной операции. Пример унарной операции – отрицание значения. Чтобы создать унарное выражение, инвертирующее значение, используется статический метод Negate класса Expression.

Expressions.UnaryExpression negation = Expressions.Expression.Negate(parameter);

Для создания объекта UnaryExpression также можно использовать следующие методы:

* Метод ArrayLength создает выражение, возвращающее длину одномерного массива, передаваемого как параметр.
* Метод Convert создает выражение, преобразующее объект, первый параметр которого указывает на тип, который задается с помощью объекта System.Type в качестве второго параметра.
* Метод ConvertChecked создает выражение, преобразуюшее объект, первый параметр которого указывает на тип, который задается с помощью объекта System.Type в качестве второго параметра, и выполняет проверку переполнения преобразования
* Метод Negate создает выражение, которое инвертирует значение выражения.
* Метод NegateChecked создает выражение, которое инвертирует значение выражения и проверяет числовое переполнение.
* Метод Not создает выражение, которое выполняет побитовую операцию NOT по параметру.
* Метод Quote создает выражение, которое возвращает постоянное значение типа параметра.
* Метод TypeAs создает выражение, выполняюшее прямую ссылку или упаковку преобразования и возвращает нуль, если преобразование не удалось.
* Метод UnaryPlus создает выражение, выполняюшее операцию унарный плюс.

**Класс Expression<TDelegate>.** Лямбда-выражения содержат два основных элемента: список параметров и тело, которое может вернуть значение, основаное на вычислении, включающем в себя эти параметры. В следующем примере показано лямбда-выражение, принимающее параметр x. Тело лямбда-выражения это бинарное выражение, основанное на этом параметре.

x => x > 2

Для построения дерева выражений используется обобщенный тип Expression<TDelegate>, представляющий лямбда-выражение. Параметр типа TDelegate должнен ссылаться на делегат, соответствующий сигнатуре лямбда-выражения. В приведенном выше примере можно использовать тип Func<int,bool>, представляющий делегат, который принимает один целочисленный параметр и возвращает логическое значение.

Для построения дерева выражений используется метод Expression.Parameter, представляющий параметр лямбда-выражений. Этот метод предполагает тип параметра и имя в качестве аргументов. Затем можно построить объект BinaryExpression, который ссылается этот параметр и выполняет указанное вычисление. Наконец, можно объединить эти два дерева выражения в лямбда-выражение с помощью обобщенного метода Expression.Lambda, указывая тип делегата, на который ссылается лямбда-выражение как на параметр типа. В следующем примере показано построкемк дерева выражения для лямбда-выражения предыдущего примера.

Expression<Func<int, bool>> lambda = null;

ParameterExpression param = Expression.Parameter(typeof(int), "x");

ConstantExpression two = Expression.Constant(2, typeof(int));

BinaryExpression body = Expression.GreaterThan(param, two);

lambda = Expression.Lambda<Func<int, bool>>(body, param);

Console.WriteLine(lambda.ToString());

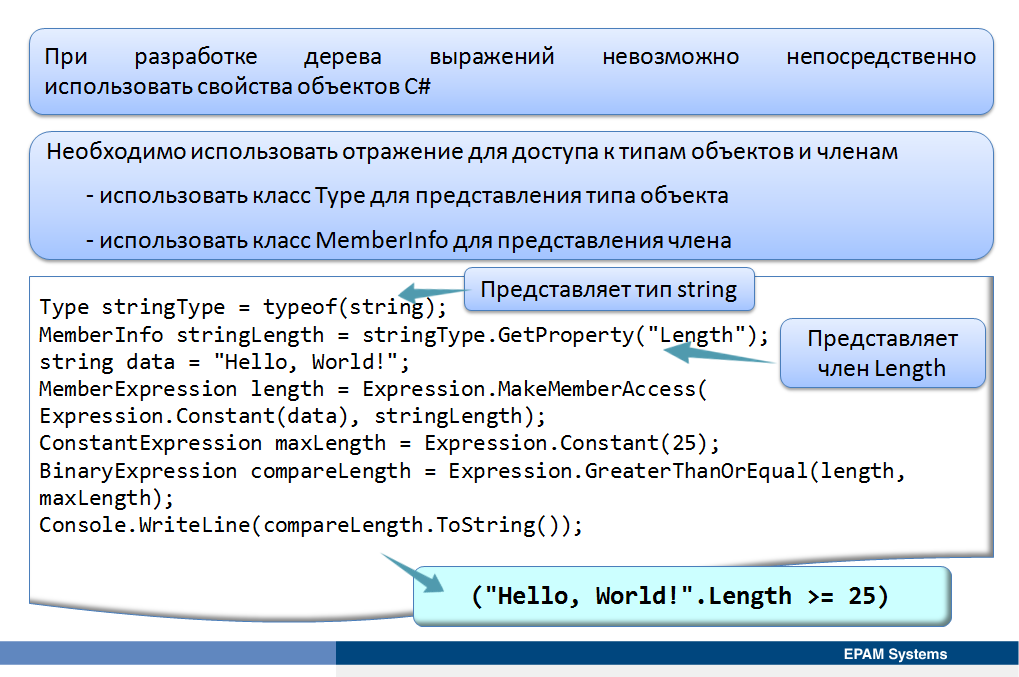
В следующем примере показан вывод, который генерирует этот код.

x => (x > 2)

http://go.microsoft.com/fwlink/?LinkId=192983

http://go.microsoft.com/fwlink/?LinkId=192984.

## Получение сведений о типах во время выполнения



C# компилятор обеспечивает безопасность типов и генерирует ошибки компиляции, если код не типизирован. При использовании деревьев выражений безопасность типа проверяется во время выполнения после построения выражения, но до вычисления. Некоторые методы Expression, такие как MakeMemberAccess, ожидают предоставления сведений о типе члена, на который ссылаются. Другие методы, такие как Expression.Constant и Expression.Parameter, ожидают предоставления информации о типе постоянной или параметре, участвующих в выражении. Для получения такого рода информации динамически можно использовать отражение.

Самым простым способом получения информации о типе является использование оператора typeof. Этот оператор ожидает тип в качестве аргумента и возвращает объект System.Type, содержащий подробные метаданные, описывающие этот тип. Общеязыковая среда выполнения (CLR) может использовать эти метаданные для выполнения проверки типов.

Использование отражения для получения информации о типе является вычислительно дорогостоящей операцией. Если нужно сослаться на тип только один раз, можо использовать ключевое слово typeof, однако, если нужно использовать тот же тип несколько раз, необходимо создать экземпляр типа Type и использовать этот экземпляр. В следующем примере показано создание экземпляра типа Type для представления типа string.

Type stringType = typeof(string);

Можно ссылаться на объект stringType, указав тип объекта ParameterExpression, как показано в следующем примере кода.

ParameterExpression param = Expression.Parameter(stringType, "data");

Класс Type позволяет ссылаться на тип, однако часто нужен доступ к членам объекта, а не к объекту. Для ссылки на член типа, можно использовать класс MemberInfo. Класс MemberInfo определен в пространстве имен System.Reflection. Чтобы создать объект MemberInfo можно использовать методы экземпляра, которые предоставляет класс Type. Есть большое количество методов, в том числе GetField и GetProperty, которые можно использовать для получения информации о полях и свойствах, предоставляемых типом. Многие из этих методов требуют указать имя члена в виде строки. Однако, если предоставляется неверное имя члена, метод будет возвращать null.

В следующем примере показано создание объекта MemberInfo для представления свойства Length объекта string. Можно использовать объект stringLength как аргумент MemberInfo метода MakeMemberAccess и построить дерево выражений, которое сравнивает длину строки с определенным значением.

Type stringType = typeof(string);

MemberInfo stringLength = stringType.GetProperty("Length");

string data = "Hello, World!";

MemberExpression length = Expression.MakeMemberAccess(

Expression.Constant(data), stringLength);

ConstantExpression maxLength = Expression.Constant(25);

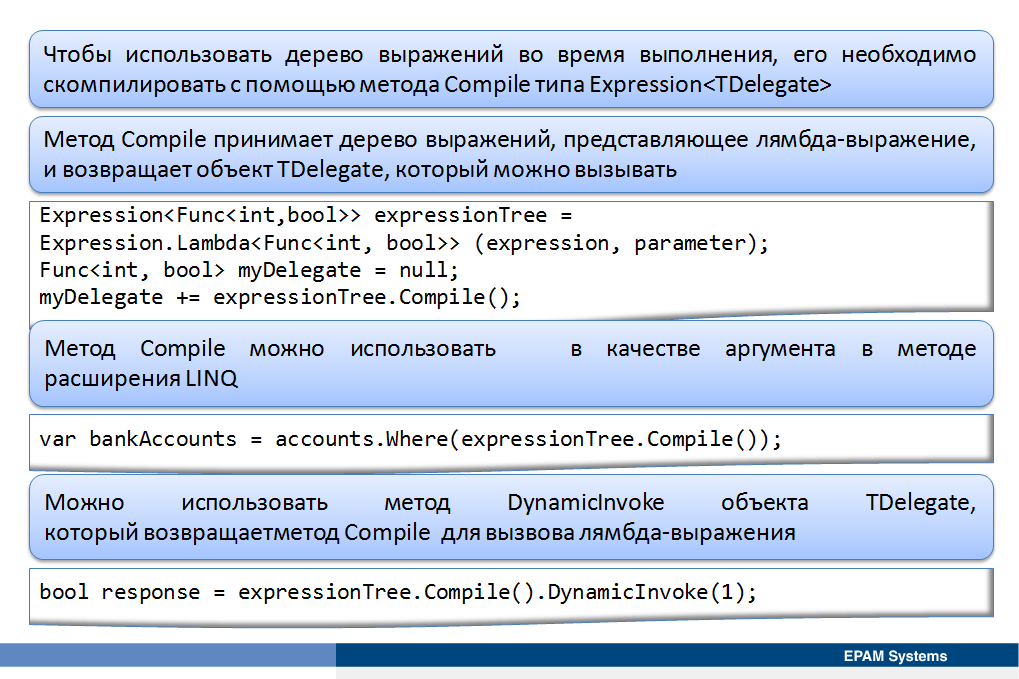
BinaryExpression compareLength = Expression.GreaterThanOrEqual(length, maxLength);

Console.WriteLine(compareLength.ToString());

Ниже показан вывод, который генерирует этот код.

("Hello, World!".Length >= 25)

## Компиляция и выполнение динамических запросов LINQ



Дерево выражений является структурой данных. Чтобы использовать дерево выражений во время выполнения, его необходимо скомпилировать. Добиться этого можно с помощью метода Compile типа Expression<TDelegate>. Метод Compile принимает дерево выражений, представляющее лямбда-выражение, и возвращает объект TDelegate, который можно вызывать. Если лямбда-выражение определяет элемент LINQ запроса, можно просто перебирать результаты, которые запрос возвращает для выполнения выражения.

Использование динамических лямбда-выражений не ограничивается LINQ запросами, построенные лямбда-выражения можно использовать динамически везде, где возможно использовать обычные, статические лямбда-выражения. Для выполнения динамического лямбда-выражения в таких ситуациях, можно использовать метод DynamicInvoke объекта TDelegate, который возвращает метод Compile.

В следующем примере показано создание динамичного LINQ запроса и результаты, полученные при его выполнении. В примере перебираются объекты TestScore, которые хранятся в коллекции List<TestScore>, и извлекаются все результаты тестов с отметкой более чем 50 и отображает имена кандидатов с этими баллами в алфавитном порядке.

class TestScore

{

public int Score { get; set; }

public string Name { get; set; }

}

class Program

{

static void Main(string[] args)

{

List<TestScore> scores = new List<TestScore>()

{

new TestScore {Score = 90, Name = "Mike"},

new TestScore {Score = 60, Name = "Louisa"},

new TestScore {Score = 85, Name = "Antony"},

new TestScore {Score = 100, Name = "Richard"},

new TestScore {Score = 45, Name = "Jason"},

new TestScore {Score = 35, Name = "Tom"},

new TestScore {Score = 96, Name = "Chris"},

new TestScore {Score = 26, Name = "Adam"},

new TestScore {Score = 71, Name = "Charles"},

new TestScore {Score = 91, Name = "Alison"},

new TestScore {Score = 34, Name = "John"}

};

var passes1 = scores.Where(testResult => testResult.Score > 50).

OrderBy(testResult => testResult.Name).

Select(testResult => testResult.Name);

// Run the query and display the results.

foreach (var pass in passes1)

{

Console.WriteLine(pass);

}

Console.WriteLine("==========================================");

Console.WriteLine("Compiling and Running a Dynamic LINQ Query");

Console.WriteLine("==========================================");

/\* The following code generates a LINQ query that is equivalent

to the following:

var passes = scores.Where(testResult => testResult.Score > 50).

OrderBy(testResult => testResult.Name).

Select(testResult => testResult.Name);

**Note that this query involves three lambda expressions:**

1. The lambda expression for the Where method takes a TestScore

object and returns a Boolean value.

2. The lambda expression for the OrderBy method takes a TestScore

object and returns a string (the data to order by).

3. The lambda expression for the Select method takes a TestScore

object and returns a string (the candidate names).

\*/

// Build the lambda expression for the Where method:

// testResult => testResult.Score > 50

Type testScoreType = typeof(TestScore);

ParameterExpression testResultParam = Expression.Parameter(testScoreType, "testResult");

MemberInfo scoreProperty = testScoreType.GetProperty("Score");

MemberExpression valueInScoreProperty = Expression.MakeMemberAccess(testResultParam, scoreProperty);

ConstantExpression fifty = Expression.Constant(50, typeof(int));

BinaryExpression scoreGreaterThanFifty = Expression.GreaterThan(valueInScoreProperty, fifty);

Expression<Func<TestScore, bool>> whereExpression =

Expression<Func<TestScore, bool>>.Lambda<Func<TestScore, bool>>(scoreGreaterThanFifty, testResultParam);

// Build the lambda expression for the OrderBy method: testResult => testResult.Name

MemberInfo nameProperty = testScoreType.GetProperty("Name");

MemberExpression valueInNameProperty = Expression.MakeMemberAccess(testResultParam, nameProperty);

Expression<Func<TestScore, string>> orderByExpression =

Expression<Func<TestScore, string>>.Lambda<Func<TestScore, string>>(valueInNameProperty, testResultParam);

// Build the lambda expression for the Select method: testResult => testResult.Name

Expression<Func<TestScore, string>> selectExpression =

Expression<Func<TestScore, string>>.Lambda<Func<TestScore, string>>(valueInNameProperty, testResultParam);

// Compile the lambda expressions, starting with the Where expression.

IEnumerable<TestScore> passingScores = scores.Where(whereExpression.Compile());

// Now append the OrderBy expression.

passingScores = passingScores.OrderBy(orderByExpression.Compile());

// Finally, add the Select expression.

IEnumerable<string> passes = passingScores.Select(selectExpression.Compile());

// Run the query and display the results.

foreach (var pass in passes)

{

Console.WriteLine(pass);

}

Console.ReadLine();

}

}

http://go.microsoft.com/fwlink/?LinkId=192985

## Демонстрация: Построение динамических запросов и выражений LINQ



1. В модуле будут описаны некоторые из наиболее часто используемых типов выражений и статических методов-фабрик, которые используются для их построения. В пространстве имен System.Linq.Expressions существует множество других доступных типов выражений. [↑](#footnote-ref-1)