Частное учреждение образования

«Колледж бизнеса и права»

|  |
| --- |
| УТВЕРЖДАЮ  Ведущий методист колледжа  \_\_\_\_\_\_\_\_\_\_\_\_\_Е.В.Паскал  «\_\_\_ » \_\_\_\_\_\_\_\_\_ 20\_\_ |

|  |  |
| --- | --- |
| Специальность: 2-40 01 01 «Программное обеспечение информационных технологий» | Учебная практика по программированию |

**Практическая работа №14**

**Инструкционно-технологическая карта**

**Тема**:LINQ

* Основы LINQ. Фильтрация выборки и проекция. Сортировка. Работа со множествами (2 часа)
* Отложенное и немедленное выполнение LINQ (2 часа)
* Делегаты и анонимные методы в запросах LINQ (2 часа)

**Цель**: Приобрести навыки работы с запросами LINQ. Научиться составлять запросы LINQ отложенного и немедленного выполнения. Развить умения по применению делегатов и анонимных методах в запросах LINQ

**Время выполнения**: 6 часов

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Ознакомиться с теоретическими сведениями.
2. Выполнить рассмотренные в теоретическом материале примеры.
3. Получить у преподавателя вариант индивидуального задания и выполнить его.
4. Оформить решение задания в отчет.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

**Основы LINQ**

LINQ (Language-Integrated Query) представляет простой и удобный язык запросов к источнику данных. В качестве источника данных может выступать объект, реализующий интерфейс IEnumerable (например, стандартные коллекции, массивы), набор данных DataSet, документ XML. Хотя все эти объекты разноплановые, но LINQ позволяет применить ко все один и тот же подход для выборки данных.

В чем же удобство LINQ? Посмотрим на простейшем примере. Выберем из массива строки, начинающиеся на определенную букву и отсортируем полученный список:

string[] teams = {"Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона"};

var selectedTeams2 = new List<string>();

foreach(string s in teams)

{

if (s.ToUpper().StartsWith("Б"))

selectedTeams2.Add(s);

}

selectedTeams2.Sort();

foreach (string s in selectedTeams2)

Console.WriteLine(s);

Теперь проведем те же действия с помощью LINQ:

string[] teams = {"Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона"};

var selectedTeams = from t in teams // определяем каждый объект из teams как t

where t.ToUpper().StartsWith("Б") //фильтрация по критерию

orderby t // упорядочиваем по возрастанию

select t; // выбираем объект

foreach (string s in selectedTeams)

Console.WriteLine(s);

Чтобы использовать функциональность LINQ, убедимся, что в файле подключено пространство имен System.LINQ.

Итак, код стал меньше и проще. В принципе все выражение можно было бы записать в одну строку: var selectedTeams = from t in teams where t.ToUpper().StartsWith("Б") orderby t select t. Но для более понятной логической разбивки я поместил каждое отдельное подвыражение на отдельной строке.

Простейшее определение запроса LINQ выглядит следующим образом:

from переменная in набор\_объектов

select переменная;

Итак, что делает этот запрос LINQ? Выражение from t in teams проходит по всем элементам массива teams и определяет каждый элемент как t. Используя переменную t мы можем проводить над ней разные операции.

Несмотря на то, что мы не указываем тип переменной t, выражения LINQ являются строго типизированными. То есть среда автоматически распознает, что набор teams состоит из объектов string, поэтому переменная t будет рассматриваться в качестве строки.

Далее с помощью оператора where проводится фильтрация объектов, и если объект соответствует критерию (в данном случае начальная буква должна быть "Б"), то этот объект передается дальше.

Оператор orderby упорядочивает по возрастанию, то есть сортирует выбранные объекты.

Оператор select передает выбранные значения в результирующую выборку, которая возвращается LINQ-выражением.

В данном случае результатом выражения LINQ является объект IEnumerable<T>. Нередко результирующая выборка определяется с помощью ключевого слова var, тогда компилятор на этапе компиляции сам выводит тип.

**Методы расширения LINQ**

Кроме стандартного синтаксиса from .. in .. select для создания запроса LINQ мы можем применять специальные методы расширения, которые определены в интерфейсе IEnumerable. Как правило, эти методы реализуют ту же функциональность, что и операторы LINQ типа where или orderby. Например:

string[] teams = { "Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона" };

var selectedTeams = teams.Where(t=>t.ToUpper().StartsWith("Б")).OrderBy(t => t);

foreach (string s in selectedTeams)

Console.WriteLine(s);

Запрос teams.Where(t=>t.ToUpper().StartsWith("Б")).OrderBy(t => t) будет аналогичен предыдущему. Он состоит из цепочки методов Where и OrderBy. В качестве аргумента эти методы принимают делегат или лямбда-выражение.

Не каждый метод расширения имеет аналог среди операторов LINQ, но в этом случае можно сочетать оба подхода. Например, используем стандартный синтаксис linq и метод расширения Count(), возвращающий количество элементов в выборке:

int number = (from t in teams where t.ToUpper().StartsWith("Б") select t).Count();

**Фильтрация**

Для выбора элементов из некоторого набора по условию используется метод Where. Например, выберем все четные элементы, которые больше 10:

int[] numbers = { 1, 2, 3, 4, 10, 34, 55, 66, 77, 88 };

IEnumerable<int> evens = from i in numbers

where i%2==0 && i>10

select i;

foreach (int i in evens)

Console.WriteLine(i);

Тот же запрос с помощью метода расширения:

IEnumerable<int> evens = numbers.Where(i => i % 2 == 0 && i > 10);

Если выражение в методе Where для определенного элемента будет равно true (в данном случае выражение i % 2 == 0 && i > 10), то данный элемент попадает в результирующую выборку.

**Проекция**

Проекция позволяет спроектировать из текущего типа выборки какой-то другой тип. Для проекции используется оператор select. Допустим, у нас есть набор объектов следующего класса, представляющего пользователя:

class User

{

public string Name { get;set; }

public int Age { get; set; }

}

Но нам нужен не весь объект, а только его свойство Name:

List<User> users = new List<User>();

users.Add(new User { Name = "Sam", Age = 43 });

users.Add(new User { Name = "Tom", Age = 33 });

var names = from u in users select u.Name;

foreach (string n in names)

Console.WriteLine(n);

Результат выражения LINQ будет представлять набор строк, поскольку выражение select u.Name выбирают в результирующую выборку только значения свойства Name.

Аналогично можно создать объекты другого типа, в том числе анонимного:

List<User> users = new List<User>();

users.Add(new User { Name = "Sam", Age = 43 });

users.Add(new User { Name = "Tom", Age = 33 });

var items = from u in users

select new

{

FirstName = u.Name,

DateOfBirth = DateTime.Now.Year - u.Age

};

foreach (var n in items)

Console.WriteLine("{0} - {1}", n.FirstName, n.DateOfBirth);

Здесь оператор select создает объект анонимного типа, используя текущий объект User. И теперь результат будет содержать набор объектов данного анонимного типа, в котором определены два свойства: FirstName и DateOfBirth. В качестве альтернативы мы могли бы использовать метод расширения Select():

// выборка имен

var names = users.Select(u => u.Name);

// выборка объектов анонимного типа

var items = users.Select(u => new

{

FirstName = u.Name,

DateOfBirth = DateTime.Now.Year - u.Age

});

**Сортировка**

Для сортировки набора данных используется оператор orderby:

int[] numbers = { 3, 12, 4, 10, 34, 20, 55, -66, 77, 88, 4 };

var orderedNumbers = from i in numbers

orderby i

select i;

foreach (int i in orderedNumbers)

Console.WriteLine(i);

Оператор orderby принимает критерий сортировки. В данном случае в качестве критерия выступает само число.

Возьмем посложнее пример. Допустим, надо отсортировать выборку сложных объектов. Тогда в качестве критерия мы можем указать свойство класса объекта:

List<User> users = new List<User>();

users.Add(new User { Name = "Sam", Age = 43 });

var sortedUsers = from u in users

orderby u.Name

select u;

foreach (User u in sortedUsers)

Console.WriteLine(u.Name);

По умолчанию оператор orderby производит сортировку по возрастанию. Однако с помощью ключевых слов ascending (сортировка по возрастанию) и descending (сортировка по убыванию) можно явным образом указать направление сортировки:

var sortedUsers = from u in users

orderby u.Name descending

select u;

Вместо оператора orderby можно использовать методы расширения OrderBy:

int[] numbers = { 3, 12, 4, 10, 34, 20, 55, -66, 77, 88, 4 };

IEnumerable<int> sortedNumbers = numbers.OrderBy(i=>i);

List<User> users = new List<User>()

{

new User { Name = "Tom", Age = 23 },

new User { Name = "Sam", Age = 43 }

};

var sortedUsers = users.OrderBy(u=>u.Name);

Метод OrderBy()сортирует по возрастанию. Для сортировки по убыванию используется метод :

var sortedUsers = users.OrderByDescending(u=>u.Name);

В наборах сложных объектов иногда встает ситуация, когда надо отсортировать не по одному, а сразу по нескольким полям. Для этого служат методы ThenBy()(для сортировки по возрастанию) и ThenByDescending() (для сортировки по возрастанию):

List<User> users = new List<User>()

{

new User { Name = "Tom", Age = 35 },

new User { Name = "Sam", Age = 43 },

new User { Name = "Tom", Age = 31 }

};

var result = users.OrderBy(u => u.Name).ThenBy(u => u.Age);

foreach (User u in result)

Console.WriteLine("Имя: {0} Возраст: {1}", u.Name, u.Age);

**Работа с множествами**

Кроме методов выборки LINQ имеет несколько методов для работы с множествами: разность, объединение и пересечение.

**Разность множеств**

С помощью метода Except можно получить разность двух множеств:

string[] soft = { "Microsoft", "Google", "Apple"};

string[] hard = { "Apple", "IBM", "Samsung"};

// разность множеств

var result = soft.Except(hard);

foreach (string s in result)

Console.WriteLine(s);

В данном случае из массива soft убираются все элементы, которые есть в массиве hard. Результатом операции будут два элемента:

Microsoft

Google

**Пересечение множеств**

Для получения пересечения множеств, то есть общих для обоих наборов элементов, применяется метод Intersect:

string[] soft = { "Microsoft", "Google", "Apple"};

string[] hard = { "Apple", "IBM", "Samsung"};

// пересечение множеств

var result = soft.Intersect(hard);

foreach (string s in result)

Console.WriteLine(s);

Так как оба набора имеют только один общий элемент, то соответственно только он и попадет в результирующую выборку:

Apple

**Объединение множеств**

Для объединения двух множеств используется метод Union. Его результатом является новый набор, в котором имеются элементы, как из одного, так и из второго множества. Повторяющиеся элементы добавляются в результат только один раз:

string[] soft = { "Microsoft", "Google", "Apple"};

string[] hard = { "Apple", "IBM", "Samsung"};

// объединение множеств

var result = soft.Union(hard);

foreach (string s in result)

Console.WriteLine(s);

Результатом операции будет следующий набор:

Microsoft

Google

Apple

IBM

Samsung

Если же нам нужно простое объединение двух наборов, то мы можем использовать метод Concat:

var result = soft.Concat(hard);

Те элементы, которые встречаются в обоих наборах, дублируются.

**Удаление дубликатов**

Для удаления дублей в наборе используется метод Distict:

var result = soft.Concat(hard).Distinct();

Последовательное применение методов Concat и Distinct будет подобно действию метода Union.

**Агрегатные операции. Методы Skip и Take**

К агрегатным операциям относят различные операции над выборкой, например, получение числа элементов, получение минимального, максимального и среднего значения в выборке, а также суммирование значений.

**Получение размера выборки. Метод Count**

Для получения числа элементов в выборке используется метод Count():

int[] numbers = { 1, 2, 3, 4, 10, 34, 55, 66, 77, 88 };

int size = (from i in numbers where i % 2 == 0 && i > 10 select i).Count();

Console.WriteLine(size);

Метод Count() в одной из версий также может принимать лямбда-выражение, которое устанавливает условие выборки. Поэтому мы можем в данном случае не использовать выражение Where:

int size = numbers.Count(i => i % 2 == 0 && i > 10);

Console.WriteLine(size);

**Получение суммы**

Для получения сумму значений применяется метод Sum:

int[] numbers = { 1, 2, 3, 4, 10, 34, 55, 66, 77, 88 };

List<User> users = new List<User>()

{

new User { Name = "Tom", Age = 23 },

new User { Name = "Sam", Age = 43 },

new User { Name = "Bill", Age = 35 }

};

int sum1 = numbers.Sum();

decimal sum2 = users.Sum(n => n.Age);

Метод Sum() имеет ряд перегрузок. В частности, если у нас набор сложных объектов, как в примере выше, то мы можем указать свойство, значения которого будут суммироваться: users.Sum(n => n.Age)

**Максимальное, минимальное и среднее значения**

Для нахождения минимального значения применяется метод Min(), для получения максимального - метод Max(), а для нахождения среднего значения - метод Average(). Их действие похоже на методы Sum и Count:

int[] numbers = { 1, 2, 3, 4, 10, 34, 55, 66, 77, 88 };

List<User> users = new List<User>()

{

new User { Name = "Tom", Age = 23 },

new User { Name = "Sam", Age = 43 },

new User { Name = "Bill", Age = 35 }

};

int min1 = numbers.Min();

int min2 = users.Min(n => n.Age); // минимальный возраст

int max1 = numbers.Max();

int max2 = users.Max(n => n.Age); // максимальный возраст

double avr1 = numbers.Average();

double avr2 = users.Average(n => n.Age); //средний возраст

**Методы Skip и Take**

Метод Skip() пропускает определенное количество элементов, а метод Take() извлекает определенное число элементов. Нередко данные методы применяются вместе для создания постраничного вывода. Извлечем три первых элемента:

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

var result = numbers.Take(3);

foreach (int i in result)

Console.WriteLine(i);

Выберем все элементы, кроме первых трех:

var result = numbers.Skip(3);

Совмещая оба метода, мы можем выбрать определенное количество элементов начиная с определенного элемента. Например, выберем три элемента, начиная с пятого (то есть пропустив четыре элемента):

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

var result = numbers.Skip(4).Take(3);

foreach (int i in result)

Console.WriteLine(i);

**Группировка**

Для группировки данных по определенным параметрам применяется оператор group by или метод GroupBy(). Допустим, у нас есть набор из объектов следующего типа:

class Team

{

public string Name { get; set; }

public string Country { get; set; }

}

Данный класс представляет футбольную команду, в которой определены свойства для названия команды и страны. Сгруппируем набор команд по стране:

List<Team> teams = new List<Team>()

{

new Team { Name = "Бавария", Country ="Германия" },

new Team { Name = "Реал Мадрид", Country ="Испания" },

new Team { Name = "Атлетико", Country ="Испания" },

new Team { Name = "Боруссия", Country ="Германия" },

new Team { Name = "Манчестер Сити", Country ="Англия" },

new Team { Name = "Барселона", Country ="Испания" },

new Team { Name = "Челси", Country ="Англия" }

};

var result = from t in teams

group t by t.Country;

foreach (var g in result)

{

Console.WriteLine(g.Key);

foreach(var t in g)

Console.WriteLine(t.Name);

Console.WriteLine();

}

Если в выражении LINQ последним оператором, выполняющим операции над выборкой, является group, то оператор select не применяется. Оператор group принимает критерий по которому проводится группировка: group t by t.Country - в данном случае группировка по свойству Country. Результатом оператора group является выборка, которая состоит из групп. В данном случае у нас получается три группы по числу стран. Каждая группа имеет ключ, который мы можем получить через свойство Key: g.Key. Все элементы группы можно получить с помощью дополнительной итерации. Элементы группы имеют тот же тип, что и тип объектов, которые передавались оператору group, то есть в данном случае объекты типа Team.

В итоге получим следующий вывод:

Германия

Бавария

Боруссия

Испания

Реал Мадрид

Атлетико

Барселона

Англия

Манчестер Сити

Челси

Теперь изменим запрос и получим команду и создадим из группы новый объект:

var result2 = from t in teams

group t by t.Country into g

select new { Name = g.Key, Count = g.Count() };

foreach (var c in result2)

Console.WriteLine("Страна: {0} Кол-во команд: {1}", c.Name, c.Count);

Выражение group t by t.Country into g определяет переменную g, которая будет содержать группу. С помощью этой переменной мы можем затем создать новый объект анонимного типа: select new { Name = g.Key, Count = g.Count() }.

Теперь результат запроса LINQ будет представлять набор объектов таких анонимных типов, у которых два свойства Name и Count.

Все подобные операции можно произвести с помощью метода GroupBy():

var result = teams.GroupBy(t => t.Country);

foreach (var g in result)

{

Console.WriteLine(g.Key);

foreach(var t in g)

Console.WriteLine(t.Name);

Console.WriteLine();

}

var result2 = teams.GroupBy(t => t.Country)

.Select(g => new { Name = g.Key, Count = g.Count() });

foreach (var c in result2)

Console.WriteLine("Страна: {0} Кол-во команд: {1}", c.Name, c.Count);

**Соединение. Метод Join**

Соединение в LINQ используется для объединения двух разнотипных наборов в один. Для соединения используется оператор join или метод Join(). Как правило, данная операция применяется к двум наборам, которые имеют один общий критерий. Например, у нас есть два класса:

class Player

{

public string Name { get; set; }

public string Team { get; set; }

}

class Team

{

public string Name { get; set; }

public string Country { get; set; }

}

Объекты обоих классов будет иметь один общий критерий - название команды. Соединим по этому критерию два набора этих классов:

List<Team> teams = new List<Team>()

{

new Team { Name = "Бавария", Country ="Германия" },

new Team { Name = "Барселона", Country ="Испания" }

};

List<Player> players = new List<Player>()

{

new Player {Name="Месси", Team="Барселона"},

new Player {Name="Неймар", Team="Барселона"},

new Player {Name="Роббен", Team="Бавария"}

};

var result = from pl in players

join t in teams on pl.Team equals t.Name

select new { Name = pl.Name, Team = pl.Team, Country = t.Country };

foreach (var item in result)

Console.WriteLine("{0} - {1} ({2})", item.Name, item.Team, item.Country);

С помощью выражения join t in teams on pl.Team equals t.Name объект pl из списка players соединяется с объектом t из списка teams, если значение свойства pl.Team совпадает со значением свойства t.Name. Результатом соединения будет объект анонимного типа, который будет содержать три свойства. В итоге мы получим следующий вывод:

Месси - Барселона (Испания)

Неймар - Барселона (Испания)

Роббен - Бавария (Германия)

То же самое действие можно было бы выполнить с помощью метода Join():

var result = players.Join(teams, // второй набор

p => p.Team, // свойство-селектор объекта из первого набора

t => t.Name, // свойство-селектор объекта из второго набора

(p, t) => new { Name = p.Name, Team = p.Team, Country = t.Country }); // результат

Метод Join() принимает четыре параметра:

* второй список, который соединяем с текущим;
* свойство объекта из текущего списка, по которому идет соединение;
* свойство объекта из второго списка, по которому идет соединение;
* новый объект, который получается в результате соединения.

**Отложенное и немедленное выполнение LINQ**

Есть два способа выполнения запроса LINQ: отложенное и немедленное выполнение.

При отложенном выполнении LINQ-выражение не выполняется, пока не будет произведена итерация или перебор по выборке. Рассмотрим отложенное выполнение:

string[] teams = {"Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона"};

var selectedTeams = from t in teams where t.ToUpper().StartsWith("Б") orderby t select t;

// выполнение LINQ-запроса

foreach (string s in selectedTeams)

Console.WriteLine(s);

То есть фактическое выполнение запроса происходит не в строке определения: var selectedTeams = from t..., а при переборе в цикле foreach. Чтобы более наглядно увидеть это, мы можем изменить какой-либо элемент до перебора выборки:

var selectedTeams = from t in teams where t.ToUpper().StartsWith("Б") orderby t select t;

// изменение массива после определения LINQ-запроса

teams[1] = "Ювентус";

// выполнение LINQ-запроса

foreach (string s in selectedTeams)

Console.WriteLine(s);

Теперь выборка будет содержать два элемента, а не три, так как второй элемент после изменения не будет соответствовать условию.

Если в LINQ-запросе используются методы расширения, которые возвращают результат, отличающийся от типа последовательности, например, методы ToArray<T>(), ToList<T>(), Count() и т.д.. Например:

string[] teams = {"Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона"};

// выполнение LINQ-запроса

var selectedTeams = (from t in teams

where t.ToUpper().StartsWith("Б")

orderby t select t).ToList<string>();

// изменение массива никак не затронет список selectedTeams

teams[1] = "Ювентус";

foreach (string s in selectedTeams)

Console.WriteLine(s);

С помощью метода ToList() выполняется преобразование последовательности к типу List<string>, поэтому будет применяться немедленное выполнение. Поэтому вне зависимости от того, будет мы изменять массив teams или нет, запрос список selectedTeams будет содержать три элемента.

Рассмотрим еще пример с методом Count(), который возвращает число элементов последовательности:

string[] teams = {"Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона"};

// выполнение LINQ-запроса

int i = (from t in teams

where t.ToUpper().StartsWith("Б")

orderby t select t).Count();

Console.WriteLine(i); //3

teams[1] = "Ювентус";

Console.WriteLine(i); //3

Результатом метода Count будет объект int, что также отличается от типа последовательности, поэтому сработает немедленное выполнение.

Однако мы можем изменить код таким образом, чтобы метод Count() учитывал изменения:

string[] teams = {"Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона"};

// выполнение LINQ-запроса

var selectedTeams = from t in teams

where t.ToUpper().StartsWith("Б")

orderby t select t;

Console.WriteLine(selectedTeams.Count()); //3

teams[1] = "Ювентус";

Console.WriteLine(selectedTeams.Count()); //2

**Делегаты и анонимные методы в запросах LINQ**

Хотя, как правило, в качестве параметров в методах расширения LINQ удобно использовать лямбда-выражения. Но лямбда-выражения являются сокращенной нотацией анонимных методов. И если мы обратимся к определению этих методов, то увидим, что в качестве параметра многие из них принимают делегаты типа Func<TSource, bool>, например, определение метода Where:

public static IEnumerable<TSource> Where<TSource>(

this IEnumerable<TSource> source,

Func<TSource, bool> predicate

)

Зададим параметры через делегаты:

int[] numbers = { 1, 2, 3, 4, 10, 34, 55, 66, 77, 88 };

Func<int, bool> MoreThanTen = delegate(int i) { return i > 10; };

var result = numbers.Where(MoreThanTen);

foreach (int i in result)

Console.WriteLine(i);

Так как набор элементов, к которому применяется метод Where, содержит объекты int, то в делегат в качестве параметра передается объект этого типа. Возвращаемые тип должен представлять тип bool: если true, то объект int соответствует условию и попадает в выборку.

Альтернативный поход представляет перемещение всей логики в отдельный метод:

static void Main(string[] args)

{

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

var result = numbers.Where(MoreThanTen);

foreach (int i in result)

Console.WriteLine(i);

Console.Read();

}

private static bool MoreThanTen(int i)

{

return i > 10;

}

Рассмотрим другой пример. Пусть метод Select() добавляет в выборку не текущий элемент-число, а его факториал:

static void Main(string[] args)

{

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

var result = numbers.Where(i=>i>0).Select(Factorial);

foreach (int i in result)

Console.WriteLine(i);

Console.Read();

}

static int Factorial(int x)

{

int result = 1;

for (int i = 1; i <= x; i++)

result \*= i;

return result;

}

Метод Select в качестве параметра принимает тип Func<TSource, TResult> selector. Так как у нас набор объектов int, то входным параметром делегата также будет объект типа int. В качестве типа выходного параметра выберем int, так как факториал числа - это целочисленное значение.

**Делегаты и анонимные методы в запросах LINQ**

Хотя, как правило, в качестве параметров в методах расширения LINQ удобно использовать лямбда-выражения. Но лямбда-выражения являются сокращенной нотацией анонимных методов. И если мы обратимся к определению этих методов, то увидим, что в качестве параметра многие из них принимают делегаты типа Func<TSource, bool>, например, определение метода Where:

public static IEnumerable<TSource> Where<TSource>(

this IEnumerable<TSource> source,

Func<TSource, bool> predicate

)

Зададим параметры через делегаты:

int[] numbers = { 1, 2, 3, 4, 10, 34, 55, 66, 77, 88 };

Func<int, bool> MoreThanTen = delegate(int i) { return i > 10; };

var result = numbers.Where(MoreThanTen);

foreach (int i in result)

Console.WriteLine(i);

Так как набор элементов, к которому применяется метод Where, содержит объекты int, то в делегат в качестве параметра передается объект этого типа. Возвращаемые тип должен представлять тип bool: если true, то объект int соответствует условию и попадает в выборку.

Альтернативный поход представляет перемещение всей логики в отдельный метод:

static void Main(string[] args)

{

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

var result = numbers.Where(MoreThanTen);

foreach (int i in result)

Console.WriteLine(i);

Console.Read();

}

private static bool MoreThanTen(int i)

{

return i > 10;

}

Рассмотрим другой пример. Пусть метод Select() добавляет в выборку не текущий элемент-число, а его факториал:

static void Main(string[] args)

{

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

var result = numbers.Where(i=>i>0).Select(Factorial);

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

Console.Read();

}

static int Factorial(int x)

{

int result = 1;

for (int i = 1; i <= x; i++) result \*= i;

return result;

}

Метод Select в качестве параметра принимает тип Func<TSource, TResult> selector. Так как у нас набор объектов int, то входным параметром делегата также будет объект типа int. В качестве типа выходного параметра выберем int, так как факториал числа - это целочисленное значение.

ИНФОРМАЦИОННЫЕ ИСТОЧНИКИ

1. Албахари, Дж. LINQ. Карманный справочник / Дж. Албахари, Б. Албахари. – СПб.: БХВ-Петербург, 2009. – 240 с.: ил.
2. Албахари, Дж. C# 7.0. Справочник. Полное описание языка / Дж. Албахари, Б. Албахари. – СПб. : ООО «Альфа-книга», 2018. – 1024 с. : ил.
3. Прайс, Марк. C# 8 и .NET Core. Разработка и оптимизация / Марк Прайс. – СПб.: Питер, 2021. – 816 с.: ил.
4. Раттц –мл., Джозеф С. LINQ: язык интегрированных запросов в C# 2008 для профессионалов / Джозеф С. Раттц-мл. – М.: ООО «И.Д. Вильямс», 2008. – 560 с.: ил.
5. Стиллмен, Эндрю. Изучаем C# / Эндрю Стиллмен, Дженнифер Грин. – 3-е изд. – СПб.: Питер, 2020. – 816 с.: ил.
6. Троелсен, Э. Язык программирования C# 7 и платформы .NET и .NET Core / Э. Троелсен, Ф. Джепикс. – 8-е изд. – СПб. : ООО «Диалектика», 2018. – 1328 с.: ил.
7. Шарп, Дж. Microsoft Visual C#. Подробное руководство / Дж. Шарп. – 8-е изд. – СПб.: Питер, 2017. – 848 с.: ил.
8. LINQ [Электронный ресурс]. – metanit.com, 2012-2021. – Режим доступа: https://metanit.com/sharp/tutorial/15.1.php.

Преподаватель Е.В. Багласова

|  |
| --- |
| Рассмотрено на заседании цикловой  комиссии программного обеспечения информационных технологий №10  Протокол №\_\_\_\_\_от \_\_\_\_\_\_\_\_\_\_\_\_\_\_  Председатель ЦК \_\_\_\_\_\_В.Ю.Михалевич |