# 17. LINQ to Objects

Любое приложение должно обрабатывать данные. Данные могут храниться в любой форме: в XML-файлах, в коллекциях, массивах , в реляционных базах данных и т.д. Набор технологий LINQ (Language INtegrated Query — язык интегрированных запросов), появившийся в .NET 3.5, предоставил удобный способ доступа к различным хранилищам данных.

На самом высоком уровне LINQ можно воспринимать как строго типизированный язык запросов, встроенный непосредственно в грамматику самого языка С#. Используя LINQ, можно строить любое количество выражений, которые выглядят и ведут себя подобно SQL-запросам к базе данных. Однако запрос LINQ может применяться к любому числу хранилищ данных, включая хранилища, которые не имеют ничего общего с реляционными базами данных.

Строго говоря, "LINQ" — это термин, описывающий общий подход к доступу к данным. В зависимости от места использования запросов LINQ, существуют разновидности LINQ, которые перечислены ниже.

* LINQ to Objects. Эта разновидность позволяет применять запросы LINQ к массивам и коллекциям.
* LINQ to XML. Эта разновидность позволяет применять LINQ для манипулирования и опроса документов XML.
* LINQ to DataSet. Эта разновидность позволяет применять запросы LINQ к объектам DataSet из ADO.NET.
* LINQ to Entities. Эта разновидность позволяет применять запросы LINQ внутри API-интерфейса ADO.NET Entity Framework (EF).
* Parallel LINQ (он же PLINQ). Эта разновидность позволяет выполнять параллельную обработку данных, возвращенных запросом LINQ.

Для того чтобы работать с LINQ to Objects, потребуется обеспечить, чтобы в каждом файле кода С#, содержащем запросы LINQ, импортировалось пространство имен System.Linq;

Рассмотрим ключевые конструкции С#, которые обеспечили возможность существования LINQ.

Язык С# использует следующие связанные с LINQ средства:

* неявно типизированные локальные переменные;
* синтаксис инициализации объектов и коллекций;
* лямбда-выражения;
* расширяющие методы;
* анонимные типы.

## Неявная типизация локальных переменных

Ключевое слово **var** позволяет определять локальную переменную без явной спецификации лежащего в основе типа данных. Тем не менее, такая переменная будет строго типизированной, поскольку компилятор определит ее корректный тип данных исходя из начального присваивания.

// Неявно типизированные локальные переменные.

var mylnt = 0;

var myBool = true;

var myString = "Time, marches on...";

Это средство языка очень удобно и зачастую обязательно, когда используется LINQ.

## Синтаксис инициализации объектов и коллекций

При инициализации объектов можно создать переменную типа класса или структуры и установить любое количество ее общедоступных свойств за один прием.

List<Rectangle> myListOfRects = new List<Rectangle>

{

new Rectangle {TopLeft = new Point { X = 10, Y = 10 },

BottomRight = new Point { X = 200, Y = 200}},

new Rectangle {TopLeft = new Point { X = 2, Y = 2 },

BottomRight = new Point { X = 100, Y = 100}},

new Rectangle {TopLeft = new Point { X = 5, Y = 5 },

BottomRight = new Point { X = 90, Y = 75}}

};

Этот синтаксис в сочетании с неявной типизацией локальных переменных позволяет объявлять анонимный тип, что очень удобно для создании проекций LINQ.

## Лямбда-выражения

Лямбда-выражение может быть использовано в любой момент, когда вызывается метод, который требует строго типизированного делегата в качестве аргумента. Лямбда-выражения упрощают работу с делегатами.

Лямбда-выражения могут быть описаны следующим образом:

АргументыДляОбработки => ОператорыДляИхОбработки

**Пример.**

static void Main(string[] args)

{

// Создать список целых чисел.

List<int> list = new List<int>();

list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

// Лямбда-выражение С#.

**List<int> evenNumbers = list.FindAll (l => (l % 2) == 0) ;**

// Вывод на консоль четных чисел.

Console .WriteLine ("Here are your even numbers:");

foreach (int evenNumber in evenNumbers)

{

Console.Write("{0}\t", evenNumber);

}

Console.WriteLine();

}

Метод FindAll() – принимает предикат Predicate<T> и возвращает новый список с элементами исходного списка, соответствующими предикату. Predicate<T> - это делегат, предоставляющий метод, в котором задан набор критериев, он позволяет определить, удовлетворяет ли этим критериям заданный объект. Предикат задается с помощью ламбда выражения.

## Расширяющие методы

Расширяющие методы позволяют существующим скомпилированным типам (классам, структурам или реализациям интерфейсов) получать новую функциональность без необходимости в непосредственном изменении расширяемого типа. В качестве первого параметра такого метода используется ключевое слово **this**, которое и помечает расширяемый тип. Кроме того, расширяющие методы должны всегда определяться внутри статического класса, а потому объявляться с использованием ключевого слова static.

Пример.

**namespace CustomExtensions**

{

//Extension methods must be defined in a static class

public **static** class StringExtension

{

// Определение расширяющего метода для System.String.

public **static** int WordCount(**this** string str)

{

return str.Split(new char[] { ' ', '.', '?' }, StringSplitOptions.RemoveEmptyEntries).Length;

}

}

static class ObjectExtensions

{

// Определение расширяющего метода для System.Object.

public static void DisplayName(this object obj)

{

Console.WriteLine( obj.GetType().Name);

}

}

}

namespace \_17\_1\_4

{

**using CustomExtensions;**

class Program

{

static void Main(string[] args)

{

string s = "The quick brown fox jumped over the lazy dog."

**int i = s.WordCount();**

Console.WriteLine("Word count of s is {0}", i);

int myInt = 12345678;

**myInt.DisplayName();**

}

}

}

## Анонимные типы

Анонимный тип определяется с использованием ключевого слова var в сочетании с синтаксисом инициализации объекта.

// Создать анонимный тип, представляющий автомобиль.

var myCar = new { Color = "Bright Pink", Make = "Saab", CurrentSpeed =55 };

// Вывести на консоль цвет и производителя.

Console.WriteLine("My car is a {0} {1}.", myCar.Color, myCar.Make);

## Применение запросов LINQ к элементарным массивам

**Задача.** Дан массив строк, вывести в алфавитном порядке строки, содержащие пробелы.

1. **Решение без LINQ**

static void QueryWithoutLINQ()

{

// Предположим, что имеется массив строк,

string[] currentVideoGames = { "Morrowind", "Uncharted 2", "Fallout 3", "Daxter", "System Shock 2" };

string[] gamesWithSpaces = new string[5];

for (int i = 0; i < currentVideoGames.Length; i++)

{

if (currentVideoGames[i].Contains(" "))

gamesWithSpaces[i] = currentVideoGames[i];

}

// Отсортировать набор.

Array.Sort(gamesWithSpaces);

// Вывести на консоль результат,

foreach (string s in gamesWithSpaces)

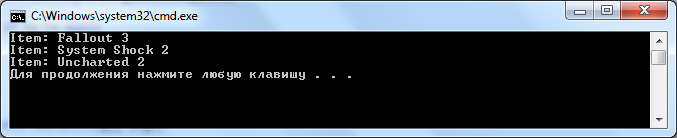
{

if (s != null)

Console.WriteLine("Item: {0}", s);

}

}



В простейшем виде каждый запрос LINQ строится из операций from, in и select. Ниже показан базовый шаблон, которому нужно следовать:

**var** результат = **from** сопоставляемыйЭлемент **in** контейнер

**select** сопоставляемыйЭлемент;

Таблица Операции запросов LINQ

|  |  |
| --- | --- |
| Операции запросов | Назначение |
| from, in | Используется для определения любого запроса |
| where | Используется для определения ограничений о том, т.е. какие  элементы должны извлекаться из контейнера |
| select | Используется для выбора последовательности из контейнера |
| join, on, equals, into | Выполняет соединения на основе указанного ключа. |
| orderby | Позволяет упорядочить результирующий набор в порядке возрастания или убывания |
| group, by | Группирует данные по указанному значению |

1. **Решение с помощью LINQ запроса**

string[] currentVideoGames = { "Morrowind", "Uncharted 2", "Fallout 3", "Daxter", "System Shock 2"};

//Запрос для получения строк содержащих пробелы, вывести //результат в алфавитном порядке

IEnumerable<string> subset = **from** game **in** currentVideoGames **where** game.Contains(" ")

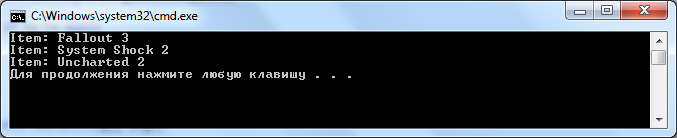
**orderby** game

**select** game;

// Вывести на консоль результаты,

foreach (string s in subset)

Console.WriteLine("Item: {0}", s);



Возвращенная последовательность хранится в переменной по имени subset, тип которой реализует обобщенную версию интерфейса IEnumerable<T>, где Т — тип string. Но реально в эту переменную могут быть записаны данные любого типа, наследуемого от IEnumerable<>.

Для получения результата запроса более удобно использовать **неявно типизированную локальную переменную**

**var** subset = from g in currentVideoGames

where g.Contains(" ")

orderby g

select g;

foreach (**var** s in subset)

Console.WriteLine("Item: {0}", s);

При получении результатов запроса LINQ всегда следует применять неявную типизацию. Однако помните, что (в большинстве случаев) реальное возвращенное значение имеет тип, реализующий интерфейс IEnumerable<T>.Какой именно тип кроется за этим (OrderedEnumerable<TElement, TKey>, WhereArrayIterator<T> и т.п.) не важно, и определять его не обязательно.

Для представления массива строк используется класс System.Array, который не реализует напрямую интерфейс IEnumerable<T>. Необходимую функциональность этого типа (а также многие другие члены, связанные с LINQ) он получает через статический тип класса System.Linq. Enumerable. В этом служебном классе определено множество обобщенных **расширяющих методов** (таких как Aggregate<T>(), First<T>, Max<T> и т.д.), которые System.Array (и другие типы) получают в свое распоряжение.

Другой важный момент, касающийся выражений запросов LINQ, состоит в том, что на самом деле они не выполняются до тех пор, пока не будет начата итерация по последовательности. Это называется **отложенным выполнением**.

int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

// Получить числа меньше 10.

var subset = from i in numbers where i < 10 select i;

// Оператор LINQ здесь выполняется!

foreach (var i in subset)

Console.WriteLine("{0} < 10", i);

Console.WriteLine();

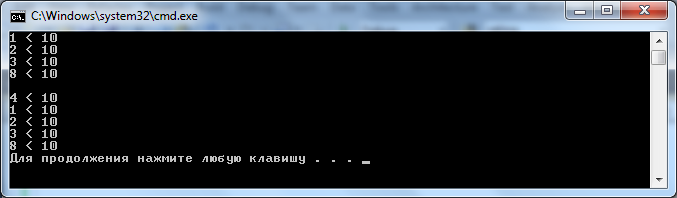
// Изменить некоторые данные в массиве.

numbers[0] = 4;

**// Оператор LINQ снова выполняется!**

foreach (var j in subset)

Console.WriteLine("{0} < 10", j);



Чтобы выполнить выражение LINQ за пределами логики итерации foreach, можно вызвать любое количество расширяющих методов, определенных типом Enumerable, таких как ToArry<T>, ToDictionary<TSource,TKey>() и ToList<T>(). Все эти методы заставляют запрос LINQ выполняться в момент их вызова (**немедленное выполнение**).

int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

// Получить данные НЕМЕДЛЕННО как int[].

int[] subsetAsIntArray =

(from l in numbers where l < 10 select l) **.ToArray<int> ();**

// Получить данные НЕМЕДЛЕННО как List<int>.

List<int> subsetAsListOfInts =

(from l in numbers where i < 10 select i) **.ToList<int> ()** ;

Кроме того, когда компилятор С# может однозначно определить параметр типа обобщенного элемента, то этот параметр можно не указывать. Таким образом, ТоАггау<Т> (или ToList<T>) можно было бы вызвать следующим образом:

int[] subsetAsIntArray =

(from i in numbers where l < 10 select i) . ToArray();

## Возврат результатов LINQ запросов

Ограничения, связанные с использованием var:

* Нельзя использовать неявную типизацию для поля класса или структуры.
* Нельзя использовать неявно типизированные переменные для определения параметров, возвращаемых значений методов.

Если есть результирующий набор, состоящий из строго типизированных данных, такой как массив строк или список List<T> объектов, можно отказаться от ключевого слова var и использовать тип IEnumerable<T> либо IEnumerable.

static IEnumerable<string> GetStringSubset ()

{

string[] colors = {"Light Red11, "Green","Yellow", "Dark Red", "Red", "Purple"};

IEnumerable<string> theRedColors = from c in colors where с.Contains("Red") select c;

return theRedColors;

}

IEnumerable<string> subset = GetStringSubset();

Этот пример работает ожидаемым образом только потому, что возвращаемое значение GetStringSubset() и запрос LINQ внутри этого метода строго типизированы.

Можно использовать для возврата немедленное выполнение и возвращать массив строк.

static string[] GetStrrigSubset()

{

. . .

return theRedColors.ToArray();

}

//Вызов метода

string[] subset = GetStrrigSubset();

В этом случае вызывающий метод не знает, что вызывается LINQ запрос.

## Применение запросов LINQ к объектам коллекций

Применение запроса LINQ к обобщенному контейнеру ничем не отличается от применения к простому массиву, поскольку LINQ to Objects может использоваться с любым типом, реализующим IEnumerable<T>.

static void GetFastCars(List<Car> myCars)

{

// Найти в контейнере List()объекты Car, у которых Speed > 55

var fastCars = from с in myCars where c. Speed > 55 select c;

foreach (var car in fastCars)

{

Console.WriteLine ("{0 } is going too fast!", car.Name);

}

}

// Найти быстрые автомобили BMW!

var fastCars = from с in myCars where c. Speed > 90 && c.Make == "BMW" select c;

## Применение запросов LINQ к необобщенным коллекциям

Операции запросов LINQ предназначены для работы с любым типом, реализующим IEnumerable<T>, поэтому для необобщенных коллекций их применять нельзя. Для необобщенных коллекций, например, ArrayList используют обобщенный расширяющий метод Enumerable.OfType<T>(). Этот метод выполняет фильтрацию элементов по заданному типу.

//необобщенная коллекция

**ArrayList myCars = new ArrayList ()** {

new Car{ Name = "Henry", Color = "Silver", Speed = 100, Make = "BMW"},

new Car{ Name = "Daisy", Color = "Tan", Speed = 90, Make = "BMW"},

new Car{ Name = "Mary", Color = "Black", Speed = 55, Make = "VW"},

new Car{ Name = "Clunker", Color = "Rust", Speed = 5, Make = "Yugo" },

new Car{ Name = "Melvin11, Color = "White", Speed = 43, Make = "Ford"}

};

// Трансформировать ArrayList в тип, совместимый с IEnumerable<T>.

**var myCarsEnum = myCars.OfType<Car>();**

// Создать выражение запроса, нацеленное на совместимый с IEnumerable<T> тип.

**var fastCars = from с in myCarsEnum where с.Speed > 55 select c;**

foreach (var car in fastCars)

{

Console.WriteLine ("{0} is going too fast!", car.PetName);

}

## Примеры запросов

1. Базовый синтаксис выборки

//исходные данные

List<Car> myCars = new List<Car>() {

new Car{ Name = "Henry", Color = "Silver", Speed = 100, Make = "BMW"},

new Car{ Name = "Daisy", Color = "Tan", Speed = 90, Make = "BMW"},

new Car{ Name = "Mary", Color = "Black", Speed = 55, Make = "VW"},

new Car{ Name = "Clunker", Color = "Rust", Speed = 5, Make = "Yugo" },

new Car{ Name = "Melvin", Color = "White", Speed = 43, Make = "Ford" }

};

//запросы

//Получить все!

**var allCars = from car in myCars select car;**

foreach (var car in allCars)

Console.WriteLine(car);

//Получить название производителей

**var makes = from car in myCars select car.Make;**

foreach (var car in makes)

Console.WriteLine(car);

1. **Получение подмножества данных**

Чтобы получить определенное подмножество из контейнера, можно воспользоваться операцией where. При этом общий шаблон запроса становится таким:

var результат = from элемент in контейнер where булевскоеВыражение select элемент;

//запросы

**var blackCars = from car in myCars where car.Color=="Black" select car ;**

foreach (var car in blackCars) Console.WriteLine(car);

1. **Проекция новых типов данных**

Новые типы данных можно также проектировать на основе существующих источников. Для этого понадобится определить оператор **select**, который динамически породит новый анонимный тип:

**var nameDesc = from car in myCars select new { car.Make, car.Color };**

foreach (var car in nameDesc)

Console.WriteLine(car.ToString());

В запросе LINQ, выполняющим проекцию нельзя узнать лежащий в ее основе тип данных, поскольку он определяется во время компиляции. В этих случаях обязательным является ключевое слово var. Кроме того, нельзя создавать методы с неявно типизированными возвращаемыми значениями. Если требуется вернуть проекцию, как результат метода, то ее нужно преобразовать в массив с помощью метода ToArray():

return nameDesc.ToArray ();

Вызвать метод, который возвращает проекцию можно следующим образом:

// должен использоваться объект System.Array, при этом

//нельзя применять синтаксис объявления массива С#

**Array** objs = GetProjectedSubset ();

foreach (object o in objs)

{

Console.WriteLine(о); // Вызывает ToString() на каждом анонимном объекте.

}

1. **Получение счетчиков с помощью Enumerable**

Для определения числа элементов, возвращаемых выражением запроса LINQ, используется расширяющий метод Count () класса Enumerable.

**int numb = (from car in myCars where car.Make == "BMW" && car.Speed >= 100 select car).Count<Car>()** ;

Console.WriteLine("Number of BMW with speed more 100 is "+numb);

1. **Изменение порядка в результирующем наборе**

Для изменения порядка элементов в результирующем наборе на противоположный используется расширяющий метод Reverse<T>() класса Enumerable.

**var blackCarsReverse = (from car in myCars where car.Color == "Black" select car).Reverse<Car>();**

foreach (var car in blackCars) Console.WriteLine(car);

1. **Сортировка**

Для сортировки элементов в подмножестве по заданному значению используется операция orderby. По умолчанию принят порядок по возрастанию, поэтому упорядочение строк производится в алфавитном порядке, числовых значений — от меньшего к большему, и т.д. Чтобы отсортировать в обратном порядке, используется операция descending.

**var subset = from car in myCars orderby car.Name select car;**

Console.WriteLine("Ordered by Name:");

foreach (var p in subset)

{

Console.WriteLine(p.ToString());

}

//по убыванию

**var subset = from car in myCars orderby car.Name descending select car;**

1. **Объединение, разность, конкатенация и пересечение данных**

Класс Enumerable поддерживает набор расширяющих методов, которые позволяют использовать два (или более) запроса LINQ в качестве основы для нахождения объединений, разностей, конкатенации и пересечений данных.

**class Program**

**{**

**class Car:IComparable**

**{**

**static string[] Names = { "Nissan", "Opel", "Toyota", "Ford", "Chevrollet", "KIA" };**

**static Random rnd = new Random();**

**public string Name { get; set; }**

**public int Speed { get; set; }**

**public Car()**

**{**

**Name = "NoName";**

**Speed = 0;**

**}**

**public Car(string name, int speed)**

**{**

**Name = name;**

**Speed = speed;**

**}**

**public override string ToString()**

**{**

**return Name + ", " + Speed.ToString();**

**}**

**public void Init(int number)**

**{**

**Name = Names[number];**

**Speed = rnd.Next(0, 10);**

**}**

**//должны быть перегружены методы Equals и GetHashCode для //проверки на равенство при выполнении методов Except, //Intersect, Union, Concat**

**public override bool Equals(object obj)**

**{**

**Car temp = (Car)obj;**

**return Name == temp.Name && Speed == temp.Speed;**

**}**

**public override int GetHashCode()**

**{**

**string temp1 = Name + Speed.ToString();**

**return temp1.GetHashCode();**

**}**

**//должен быть перегружен метод CompareTo для выполнения методов //Min и Max**

**public int CompareTo(object obj)**

**{**

**if(Speed>((Car)obj).Speed) return 1; else**

**if(Speed==((Car)obj).Speed) return 0;**

**else return -1;**

**}**

**}**

**static void Main(string[] args)**

**{**

**int listSize = 6;**

**Random rnd = new Random();**

**List<Car> list1 = new List<Car>();**

**List<Car> list2 = new List<Car>();**

**for (int i = 0; i < listSize; i++)**

**{**

**Car temp = new Car(); temp.Init(rnd.Next(0,6));**

**list1.Add(temp);**

**temp = new Car(); temp.Init(rnd.Next(0,6));**

**list2.Add(temp);**

**}**

**Console.WriteLine("Первый массив");**

**foreach (Car car in list1)**

**Console.WriteLine(car);**

**Console.WriteLine("Второй массив");**

**foreach (Car car in list2)**

**Console.WriteLine(car);**

**//Разность**

**var carDiff = (from c in list1 select c).Except(from c2 in list2 select c2);**

**Console.WriteLine("Разность множеств:");**

**foreach (var car in carDiff)**

**Console.WriteLine(car);**

**//Пересечение**

**var carlntersect = (from c in list1 select c).Intersect(from c2 in list2 select c2);**

**Console.WriteLine("Пересечение множеств:");**

**foreach (var car in carlntersect)**

**Console.WriteLine(car);**

**//Объединение**

**var carUnion = (from c in list1 select c).Union(from c2 in list2 select c2);**

**Console.WriteLine("Объединение множеств");**

**foreach (Car car in carUnion)**

**Console.WriteLine(car);**

**//Сцепление**

**var carConcat=(from c in list1 select c).Concat(from c2 in list2 select c2);**

**Console.WriteLine("Сцепление множеств");**

**foreach (Car car in carConcat)**

**Console.WriteLine(car);**

**Console.WriteLine("Сцепление множеств без дубликатов");**

**foreach (Car car in carConcat.Distinct())**

**Console.WriteLine(car);**

**}**

1. **Агрегатные операции LINQ**

Для получение среднего, максимума, минимума или суммы используются методы Мах(), Min(), Average(), Sum().

**Console.WriteLine("Max Speed={0}",(from t in list1 select t).Max());**

**Console.WriteLine("Min Speed={0}", (from t in list1 select t).Min());**

## Внутреннее представление операторов запросов LINQ

Компилятор С# на этапе компиляции транслирует все операции С# LINQ в вызовы методов класса Enumerable.

Класс Enumerable предоставляет набор методов типа static для выполнения запросов к объектам, реализующим интерфейс IEnumerable<T>.

Большинство методов Enumerable принимают в качестве аргументов делегаты. В частности, многие методы требуют обобщенного делегата по имени Func<>. Делегат Func<> представляет шаблон функции с набором аргументов и возвращаемым значением.

Поскольку множество членов System.Linq.Enumerable требуют при вызове в качестве входа делегат, можно либо вручную создать новый тип делегата и разработать для него необходимые целевые методы, воспользоваться анонимным методом С# либо определить подходящее лямбда-выражение.

1. **Построение выражений запросов с использованием операций запросов**

**string[] carNames = {"Opel Corsa","Nissan Juke", "Toyota", "Chevrollet", "Ford Focus", "KIA" };**

**var subset = from car in carNames where car.Contains(" ") orderby car select car;**

**foreach (string s in subset) Console.WriteLine("Item: {0}", s);**

В этом случае делегаты Func<> и вызовы типа Enumerable остаются вне поля зрения и внимания, поскольку работа компилятора С# состоит в выполнении необходимой трансляции. Таким образом, создание выражений LINQ с использованием различных операций запросов (from, in, where или orderby) является наиболее распространенным и простым подходом.

1. **Построение выражений запросов с использованием типа Enumerable и лямбда-выражений**

Используемые операции запросов LINQ — это на самом деле сокращенные версии вызова различных расширяющих методов, определенных в типе Enumerable.

Например

**var subset2 = carNames.Where(car => car.Contains(" ")). OrderBy(car => car).Select(car => car);**

**Console.WriteLine("Вариант 2");**

**foreach (string s in subset2)**

**Console.WriteLine("Item: {0}", s);**

Чтобы было более понятно, разобьем этот запрос на фрагменты:

1. **var carsWithSpaces = carNames.Where(car => car.Contains(" "));**
2. **var orderedGames = carsWithSpaces.OrderBy(car => car);**
3. **var subset2 = orderedGames.Select(car => car);**

1) - вызов расширяющего метода **Where ().**Класс Array получает метод от класса Enumerable.

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

Метод Enumerable.Where() требует параметра-делегата System.Func<Tl, TResult> Первый параметр делегата – это данные для обработки (массив строк в примере), второй параметр — это результат, который получается от оператора, вставленного в лямбда-выражение. Метод Where () возвращает результат типа OrderedEnumerable.

2) - Для этого результата вызывается обобщенный метод **OrderBy (),** который также принимает параметр — делегат Funс<>.С его помощью производится передача всех элементов по очереди через соответствующее лямбда-выражение. Конечным результатом вызова OrderBy () будет упорядоченная последовательность начальных данных.

3) - Производится вызов метода Select () на последовательности, возвращенной OrderBy (), который в конечном итоге вернет результирующий набор данных.

1. **Построение выражений запросов с использованием типа Enumerable и анонимных методов**

Т.к. лямбда-выражения С# — это просто сокращенная нотация вызова анонимных методов, то можно построить запрос с применением анонимных методов. Синтаксис анонимных методов позволяет заключить всю обработку, выполняемую делегатами, в одном определении метода.

**Func<string, bool> searchFilter =delegate(string car) { return car.Contains(" "); };**

**Func<string, string> itemToProcess = delegate(string s) { return s; };**

**var subset = carNames.Where(searchFilter). OrderBy(itemToProcess). Select(itemToProcess);**

Объект **searchFilter** это делегат, который принимает параметр типа string и возвращает результат типа bool.

Объект **itemToProcess** – это делегат, который принимает строку и возвращает строку, в данном случае без изменений.

Используем эти делегаты в методах:

**Where(searchFilter)** – выбирает из carNames элементы для которых результат выполнения метода делегата равен true.

Для результата полученного после применения Where() применяем метод **OrderBy(itemToProcess),** который упорядочивает строки.

Метод **Select(itemToProcess)** возвращает результат.

1. **Примеры использования метода Aggregate()**

Метод **Aggregate**() применяет к последовательности агрегатную функцию.

Синтаксис :

**public static TSource Aggregate<TSource>(**

**this IEnumerable<TSource> source,**

**Func<TSource, TSource, TSource> func**

**)**

где **TSource** – тип элементов обрабатываемой последовательности **source**,

Параметры:

**IEnumerable<T> source** – объект к которому будет применяться агрегатная функция,

**Func<TSource, TSource, TSource> func – а**грегатная функция, вызываемая для каждого элемента последовательности.

Возвращаемое значение типа **TSource** – конечное агрегатное значение.

Агрегатная функция (может предоставляться в виде лямбда-выражения) будет применяться для каждой пары элементов в коллекции от начала до конца, причем результат каждой операции будет являться входными данными следующей операции вычисления.

1. Найти сумму элементов массива, состоящего из целых чисел.

**//последовательность**

**int[] arr1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };**

**//агрегатная функция**

**int sum1 = arr1.Aggregate<int>((a, b) => a + b);**

**Console.WriteLine("sum1=" + sum1);**

здесь <int> тип элементов последовательности (массива), **(a, b) => a + b –** лямбда-выражение, которое представляет собой агрегатную функцию, суммируются элементы arr[0] и arr[1], затем к ним добавляется arr[2] и т.д.

1. Та же задача решается с помощью анонимного делегата.

**int[] arr1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };**

**Func<int,int,int> summa=delegate(int a, int b) {return a+b;};**

**int sum2 = arr1.Aggregate<int>(summa);**

**Console.WriteLine("sum2=" + sum2);**

1. Та же задача решается с помощью Linq-запроса.

**int sum3=(from num in arr1 select num).Sum();**

**Console.WriteLine("sum3=" + sum3);**

1. **Получить конкатенацию строк массива строк.**

**//исходная последовательность**

**string[] arr2 = { "Masha", "Sasha", "Pasha" };**

**//агрегатная функция в виде лямбда-выражения, последний //string – тип результата**

**Func<string, string, string> concat = (a, b) => a + " " + b;**

**//параметром является агрегатная функция**

**string concatString1 = arr2.Aggregate<string>(concat);**

**Console.WriteLine("concat1=" + concatString1);**

То же самое можно записать в виде одного оператора

**string concatString = arr2.Aggregate<string>((a, b) => a +" "+ b);**

1. Вычислить сумму длин строк, находящихся в массиве

**string[] arr2 = { "Masha", "Sasha", "Pasha" };**

**/\*агрегатная функция, которая суммирует длины строк, первый int – параметр, задающий начальное значение, последний – результат, string – строка, для которой вычисляется длина \*/**

**Func<int, string, int> length = (int s, string a) => s+ a.Length;**

**int allLength1=arr2.Aggregate<string, int>(0,length);**

**Console.WriteLine("Length = "+allLength1);**

Агрегатную функцию можно записать с помощью анонимного делегата

**Func<int, string, int> length = delegate(int s, string a) { return s + a.Length; };**

То же самое можно записать в виде одного оператора

**int sum = arr2.Aggregate<string, int>(0, (a, b) => a + b.Length);**

1. Получить конкатенацию всех строк в массиве со строкой “Example: ”

**//используется та же агрегатная функция, что и в примере 4**

**string concatString2 = arr2.Aggregate<string, string> ("Example:", concat);**

**Console.WriteLine("concat2=" + concatString2);**

1. Получить длину строки, полученной в результатеконкатенации всех строк в массиве со строкой “Example: ”

**//используются агрегатная функция из примера 4**

**int allLength2 = arr2.Aggregate<string, string, int>("Example:", concat, a => a.Length);**

**Console.WriteLine("Length = " + allLength2);**