

# Методы расширения и LINQ to Objects

Тема 17

# LINQ (Language INtegrated Query)

- Набор технологий **LINQ**, появившийся в .NET 3.5, предоставил удобный способ доступа к различным хранилищам данных (XML-файлам, коллекциям, массивам, реляционным базам данных).
- 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.
- Для того чтобы работать с LINQ to Objects, потребуется обеспечить, чтобы в каждом файле кода C#, содержащем запросы LINQ, импортировалось пространство имен **System.Linq**.

# Разновидности LINQ

- **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.

# LINQ

- Язык C# использует следующие связанные с LINQ средства:
  - неявно типизированные локальные переменные;
  - лямбда-выражения;
  - **методы расширения.**

# 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 myInt = 0;
```

```
var myBool = true;
```

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

```
var myDouble; //Error
```

# Лямбда-выражения

Лямбда выражения упрощают работу с делегатами.

```
public static int ReadIntNumbers(string message = "", Func<int, bool> condition = null)
{
    var number = 0;
    Console.WriteLine(message);
    var ok = int.TryParse(Console.ReadLine(), out number);
    if (condition != null)
        ok = ok & condition(number);
    if (!ok)
        throw new ArgumentException();
    return number;
}

//вызов
number = ReadIntNumbers("Введите целое число от -10 до 10", x => x >= -10 && x <= 10);
```

# Методы расширения

- Методы расширения 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**.

# Методы расширения

```
public static class StringExtention
{
    public static int WordCount(this string str)
    {
        return str.Split(new char[] { '.', ',', '!', ' ' },
            StringSplitOptions.RemoveEmptyEntries).Length;
    }
}
```

```
internal class Program
{
    static void Main(string[] args)
    {
        string str = "В траве сидел кузнечик";
        Console.WriteLine(str.WordCount());
    }
}
```




WordCount



# Методы расширения

```
public static class ObjectExtention
{
    public static void DisplayType(this object item)
    {
        Console.WriteLine(item.GetType().Name);
    }
}
internal class Program
{
    static void Main(string[] args)
    {
        string str = "В траве сидел кузнечик";
        str.DisplayType();
        int x = 5;
        x.DisplayType();
    }
}
```

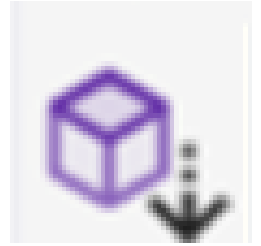


# Методы расширения для стандартных коллекций



Группа	Методы	
Агрегация	<b>Agregate, Average, Min, Max, Count, Sum</b>	
Преобразование	AsEnumerable, Cast, OfType	<b>ToArray, ToDictionary, ToList</b>
Конкатенация	Concat	
Элемент	DefaultEmpty	ElementAt, ElementAtOrDefault, <b>First</b> , FirstOrDefault, <b>Last</b> , LastOrDefault, Single, SingleOrDefault
Множество	<b>Except, Distinct, Intersect, Union</b>	
Генерация	Empty, Range, Repeat	

# Методы расширения для стандартных коллекций



Группа	Методы
Группирование	<b>GroupBy</b>
Соединение	<b>Join</b> , <b>GroupJoin</b>
Упорядочивание	<b>OrderBy</b> , <b>OrderByDescending</b> , <b>ThenBy</b> , <b>ThenByDescending</b> , <b>Reverse</b>
Проекция	<b>Select</b> , <b>SelectMany</b>
Разбиение	<b>Skip</b> , <b>SkipWhile</b> , <b>Take</b> , <b>TakeWhile</b>
Ограничение	<b>Where</b>
Квантификатор	<b>Any</b> , <b>All</b> , <b>Contains</b>
Эквивалентность	<b>SequenceEqual</b>

# Методы расширения: пример

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

```
var res1 = currentVideoGames  
    .Where(x=>x.Contains(" "))  
    .OrderBy(x=>x);
```

Метод  
расширения

Коллекция

Лямбда  
выражение

```
foreach (var item in res1)
```

```
{
```

```
    if (item != null)
```

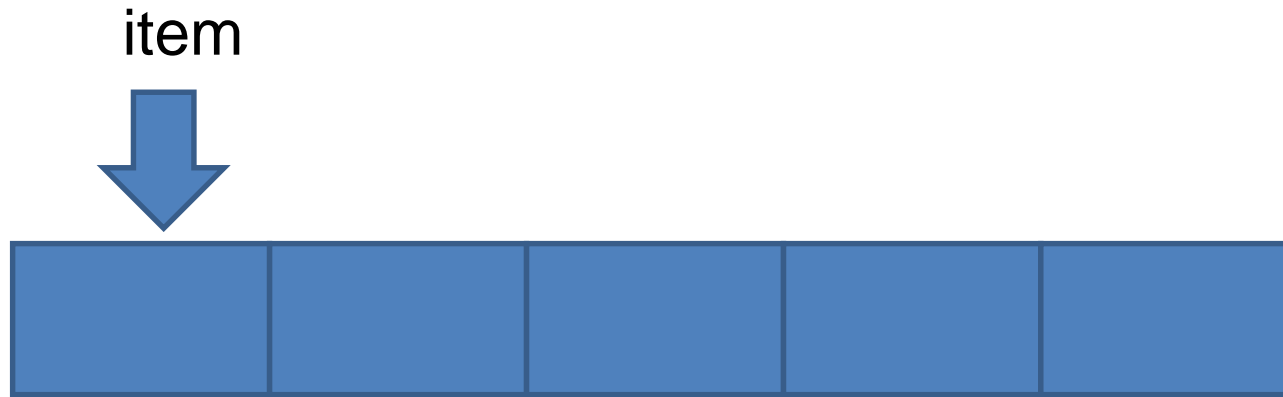
```
        Console.WriteLine(item);
```

```
}
```

Метод  
расширения

Лямбда  
выражение

# Методы расширения и запросы LINQ



Коллекция (перечислимый объект)

```
var result = collection.Select(x=>x);
```

Метод  
расширения

```
var result= from item in collection select item;
```

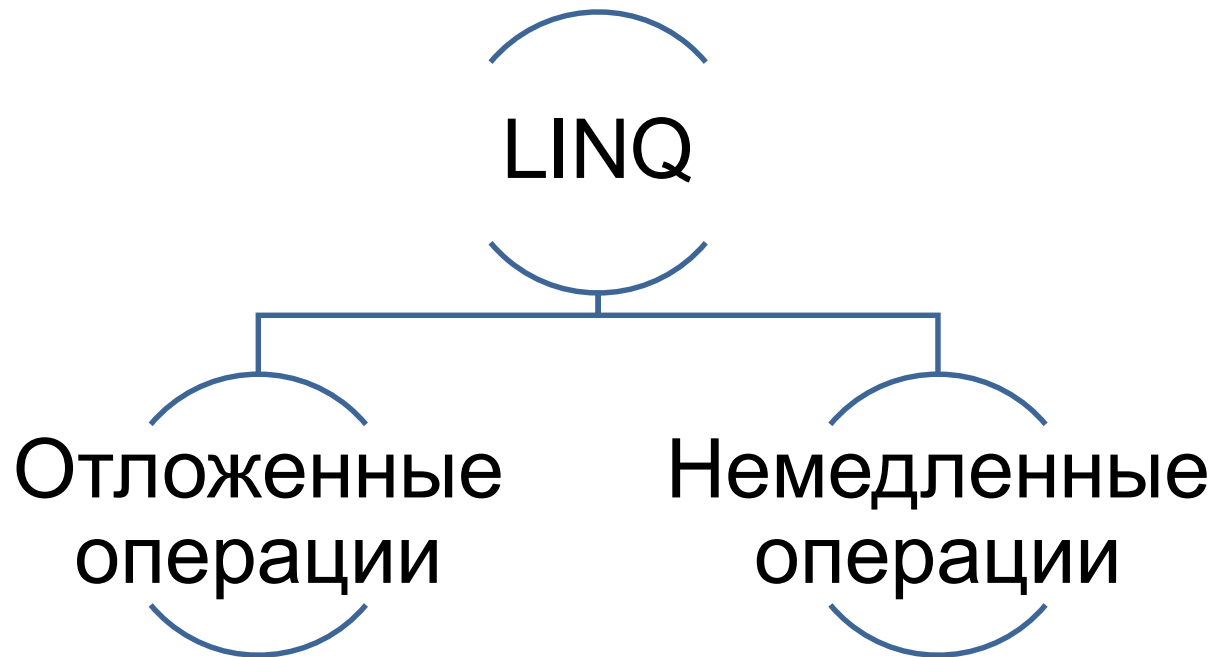
LINQ запрос

# Запросы LINQ

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

# LINQ

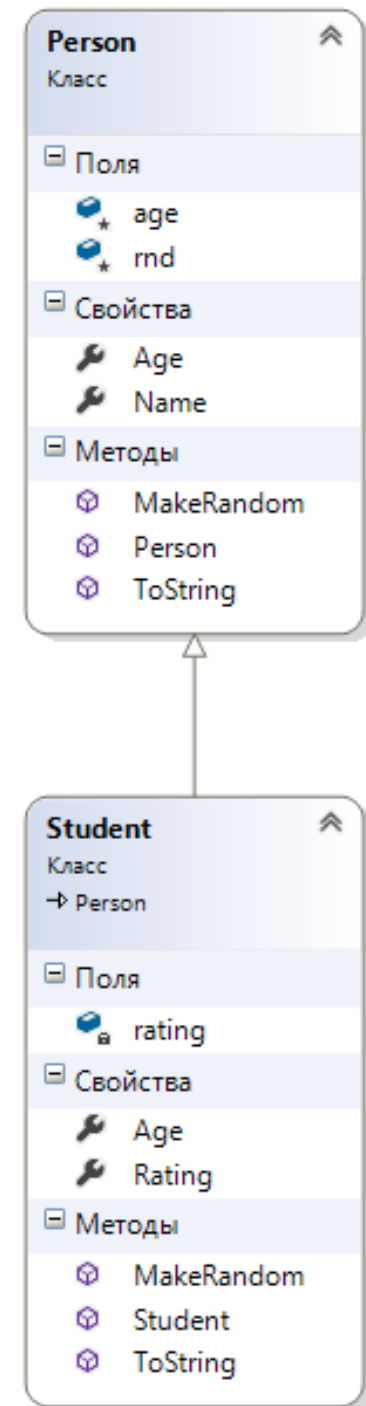
- LINQ включает в себя около 50 стандартных операций запросов, разделяемых на 2 большие группы - **отложенные операции** (выполняются не во время инициализации, а только при их вызове) и **не отложенные (немедленные) операции** (выполняются сразу).



# Пример 1

- Найти в коллекции элементов максимальный возраст.

```
List<Person> list = MakeList(10);  
foreach (Person p in list)  
    Console.WriteLine(p);  
int maxAge = list[0].Age;  
for (int i = 1; i < list.Count; i++)  
    if (list[i].Age > maxAge)  
        maxAge = list[i].Age;  
Console.WriteLine("Самый  
большой возраст:" + maxAge);
```





# Пример 1

*//метод расширения*

*maxAge = list.Max<Person>(x => x.Age);*

*Console.WriteLine(«Самый большой возраст:"  
+ maxAge);*

*//LINQ запрос*

*maxAge = (from p in list select p.Age).Max();*

*Console.WriteLine("Человек с самый большой  
возраст:" + maxAge);*

Как найти самый большой рейтинг?

# Пример 1

- Как найти самый большой рейтинг?

*double maxRating = list.Max<Person>(x => { if (x is Student) return ((Student)x).Rating; else return 0; });*

*maxRating = (from p in list where p is Student  
select ((Student)p).Rating).Max();*

- Найти средний возраст.

# Пример 1

- Найти средний возраст.

```
double avAge = list.Average(x => x.Age);  
Console.WriteLine("Средний возраст:" + avAge);
```

```
avAge = (from p in list select p.Age).Average();  
Console.WriteLine("Средний возраст:" + avAge
```

# Неявная типизация

- При получении результатов запроса LINQ следует применять неявную типизацию (var).
- В большинстве случаев реальное возвращенное значение имеет тип, реализующий интерфейс `IEnumerable<T>`.
- Какой именно тип кроется за этим не важно, и определять его не обязательно.

# Отложенное выполнение запросов

- LINQ запросы не выполняются до тех пор, пока не будет начата итерация по последовательности. Это называется **отложенным выполнением**.

```
//отложенные запросы
```

```
List<Student> collection = MakeGroup(10);
```

```
var res = from item in collection where item.Rating < 4 select item;
```

```
foreach (var item in res)
```

```
{
```

```
    Console.WriteLine(item);
```

```
}
```

```
collection.Add(new Student("Иванов Иван", 23, "РИС", 3));
```

```
Console.WriteLine("Новый результат");
```

```
foreach (var item in res)
```

```
{
```

```
    Console.WriteLine(item);
```

```
}
```

Запрос

Вывод результата

Изменение коллекции

Вывод результата

```
Денис Михайлов, 39, группа УБ, рейтинг 3
Михаил Степанов, 84, группа РИС, рейтинг 2
Денис Степанов, 62, группа ПИ, рейтинг 0
Новый результат
Денис Михайлов, 39, группа УБ, рейтинг 3
Михаил Степанов, 84, группа РИС, рейтинг 2
Денис Степанов, 62, группа ПИ, рейтинг 0
Иванов Иван, 23, группа РИС, рейтинг 3
```

# Немедленное выполнение запросов

- Чтобы выполнить запрос за пределами логики итерации foreach, можно вызвать любое количество расширяющих методов, определенных типом Enumerable, таких как ToArray<T>, ToDictionary<TSource,TKey>() и ToList<T>().

```
//немедленные запросы
List<Student> collection = MakeGroup(10);
var res = (from item in collection where item.Rating < 4 select item).ToList();
foreach (var item in res)
{
    Console.WriteLine(item);
}
collection.Add(new Student("Иванов Иван", 23, "РИС", 3));
Console.WriteLine("Новый результат");
foreach (var item in res)
{
    Console.WriteLine(item);
}
```

```
Артем Сергеев, 37, группа РИС, рейтинг 2
Степан Михайлов, 68, группа Ю, рейтинг 3
Петр Денисов, 48, группа Ю, рейтинг 3
Андрей Михайлов, 49, группа Ю, рейтинг 1
Петр Павлов, 46, группа РИС, рейтинг 2
Иван Сергеев, 78, группа Ю, рейтинг 0
Михаил Денисов, 18, группа Ю, рейтинг 3
Новый результат
Артем Сергеев, 37, группа РИС, рейтинг 2
Степан Михайлов, 68, группа Ю, рейтинг 3
Петр Денисов, 48, группа Ю, рейтинг 3
Андрей Михайлов, 49, группа Ю, рейтинг 1
Петр Павлов, 46, группа РИС, рейтинг 2
Иван Сергеев, 78, группа Ю, рейтинг 0
Михаил Денисов, 18, группа Ю, рейтинг 3
```

# Немедленное выполнение запросов

```
//немедленные запросы
List<Student> collection = MakeGroup(10);
var res = (from item in collection where item.Rating < 4 select item).Count();
Console.WriteLine($"Количество студентов с оценками ниже 4={res}");
collection.Add(new Student("Иванов Иван", 23, "РИС", 3));
Console.WriteLine("Новый результат");
Console.WriteLine($"Количество студентов с оценками ниже 4={res}");
```

```
1 Количество студентов с оценками ниже 4=5
2 Новый результат
3 Количество студентов с оценками ниже 4=5
4
```

```
//немедленные запросы
List<Student> collection = MakeGroup(10);
var res = (from item in collection where item.Rating < 4 select item).Count();
Console.WriteLine($"Количество студентов с оценками ниже 4={res}");
collection.Add(new Student("Иванов Иван", 23, "РИС", 3));
Console.WriteLine("Новый результат");
res = (from item in collection where item.Rating < 4 select item).Count();
Console.WriteLine($"Количество студентов с оценками ниже 4={res}");
```

```
Количество студентов с оценками ниже 4=3
Новый результат
Количество студентов с оценками ниже 4=4
```

# 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.

```
static IEnumerable<Student> GetExcellentStudents(List<Student> collection)
{
    return from student in collection
           where student.Rating>7
           select student;
}
```

```
var excellentStudents = GetExcellentStudents(collection);
foreach (Student student in excellentStudents)
    Console.WriteLine(student);
```



# 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.

```
static Array GetExcellentStudents(List<Student> collection)
{
    return (from student in collection
            where student.Rating>7
            select student).ToArray();
}
```

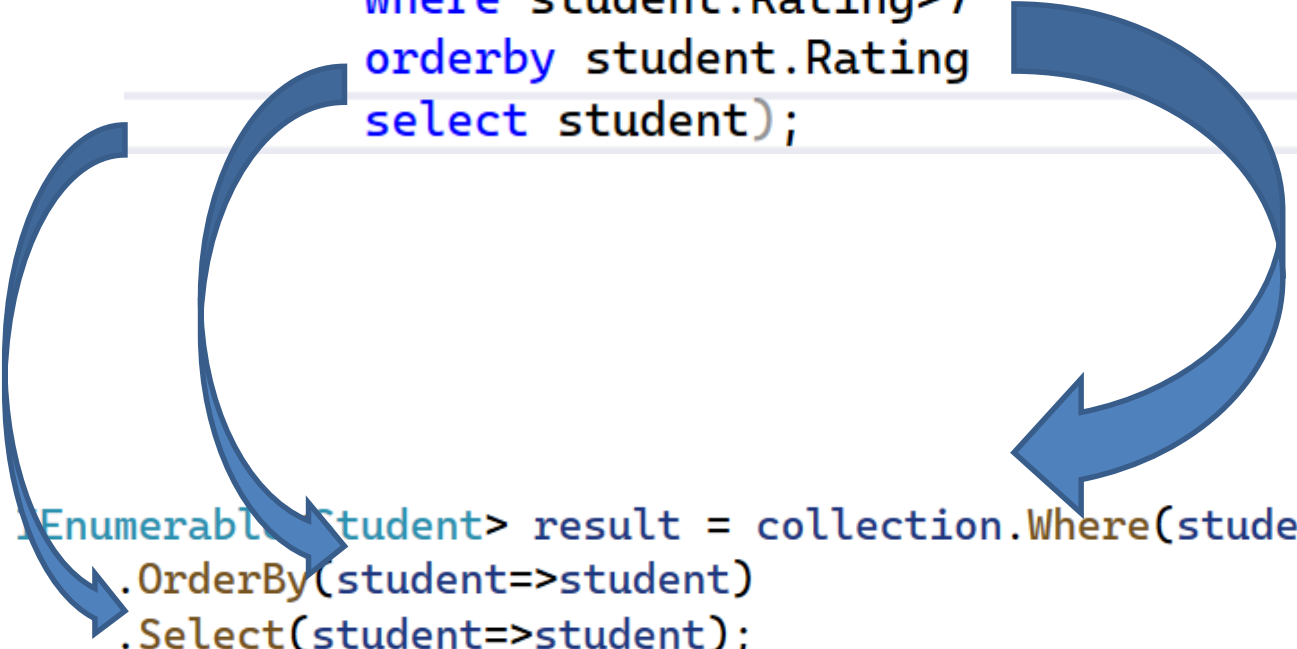
```
var excellentStudents = GetExcellentStudents(collection);
foreach (Student student in excellentStudents)
    Console.WriteLine(student);
```

# Внутреннее представление операторов запросов LINQ

- Компилятор C# на этапе компиляции транслирует все операции C# LINQ в вызовы методов класса **Enumerable**.
- Класс **Enumerable** предоставляет **набор статических методов** для выполнения запросов к объектам, реализующим интерфейс **IEnumerable<T>**.
- Большинство методов **Enumerable** принимают в качестве аргументов **делегаты** и **определены как методы расширения для IEnumerable<T>**.
- Они могут вызываться для любых объектов, реализующих **IEnumerable<T>**.
- Можно либо вручную создать новый тип делегата и разработать для него необходимые целевые методы, либо воспользоваться анонимным методом C#, либо определить подходящее лямбда-выражение.

# Пример

```
IEnumerable<Student> result=(from student in collection  
    where student.Rating>7  
    orderby student.Rating  
    select student);
```

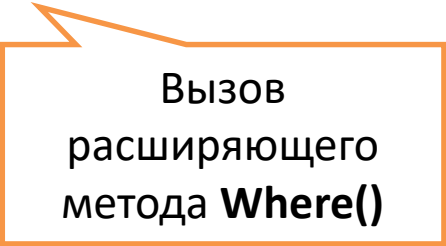


```
IEnumerable<Student> result = collection.Where(student=>student.Rating>7)  
    .OrderBy(student=>student)  
    .Select(student=>student);
```

(student=>student) – выражение тождества

# Пример

```
IEnumerable<Student> excellents = collection.Where(student => student.Rating > 7);
```



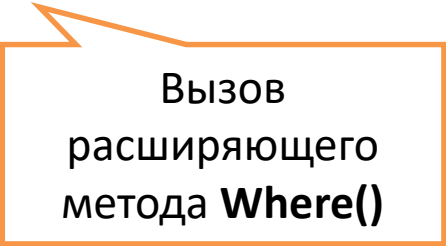
Вызов  
расширяющего  
метода **Where()**

Вызов расширяющего метода **Where()**.

- Класс List<T> получает метод Where от класса Enumerable.
- Enumerable расширяет IEnumerable<T>, List<T> реализует IEnumerable.
- Метод Enumerable.Where() принимает в качестве параметра делегат Func<>.
- Целевой метод делегата задается как лямбда-выражение

# Пример

```
IEnumerable<Student> excellents = collection.Where(student => student.Rating > 7);
```



Вызов  
расширяющего  
метода **Where()**

- Класс List<T> получает метод Where от класса Enumerable.
- Enumerable расширяет IEnumerable<T>, List<T> реализует IEnumerable.
- Метод Enumerable.Where() принимает в качестве параметра делегат Func<>.
- С его помощью производится передача всех элементов по очереди через соответствующее лямбда-выражение.
- Целевой метод делегата задается как лямбда-выражение

# Пример

```
IEnumerable<Student> orderedStudents = excellents.OrderBy(student => student);
```

Вызов  
расширяющего  
метода **OrderBy()**

- Класс List<T> получает метод OrderBy от класса Enumerable.
- Enumerable расширяет IEnumerable<T>, List<T> реализует IEnumerable.
- Метод Enumerable.OrderBy() принимает в качестве параметра делегат Func<>.
- С его помощью производится передача всех элементов по очереди через соответствующее лямбда-выражение.
- Целевой метод делегата задается как лямбда-выражение

```
(student => student)    static Student GetStudent(Student s)
                        {
                        |
                        |   return s;
                        |
                        }
```

# Пример

```
IEnumerable<Student>result=orderedStudents.Select(student => student);
```

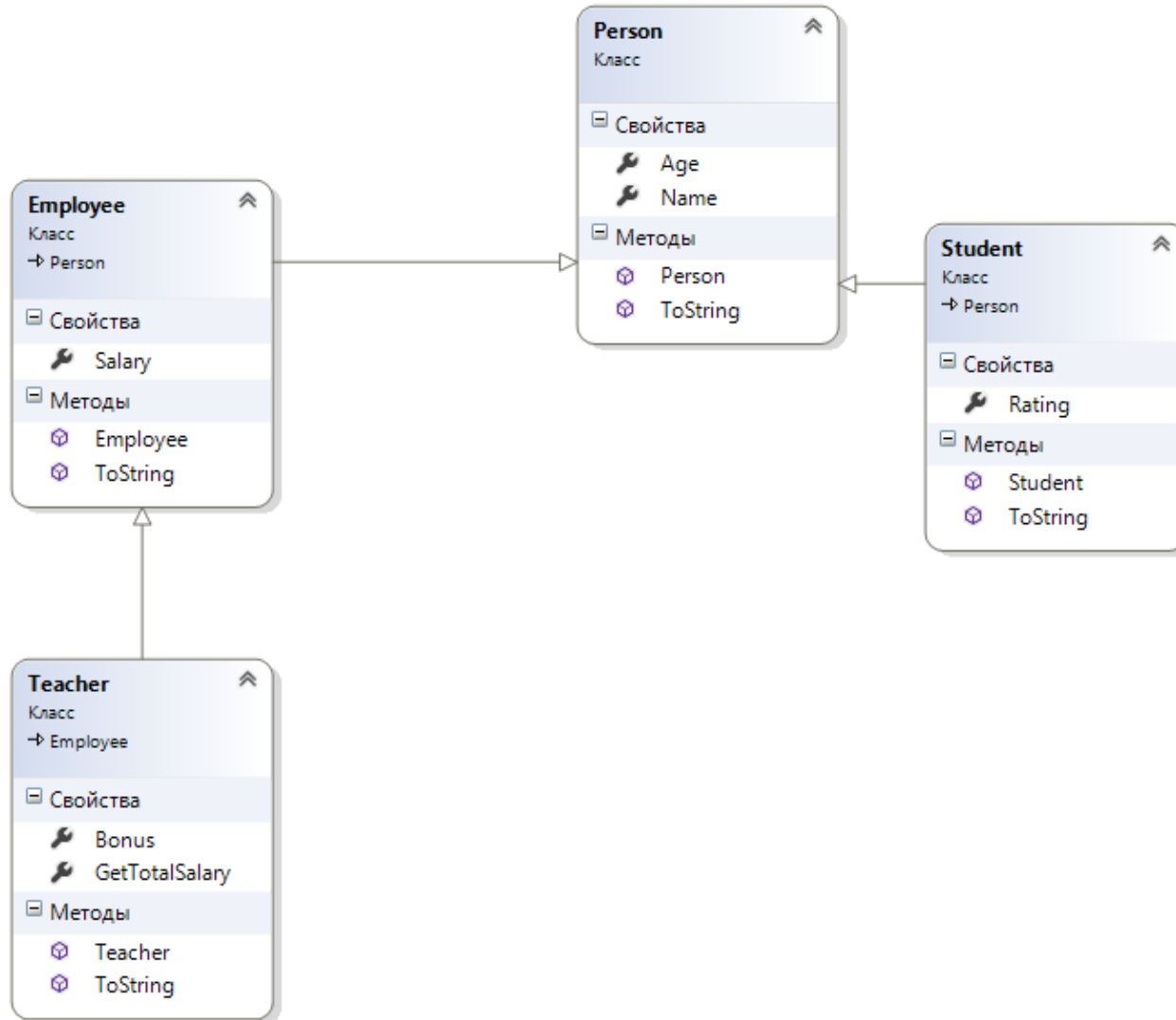
Вызов  
расширяющего  
метода **Select()**

- Класс List<T> получает метод Select() от класса Enumerable.
- Enumerable расширяет IEnumerable<T>, List<T> реализует IEnumerable.
- Метод Enumerable.Select() принимает в качестве параметра делегат Func<>.
- С его помощью производится передача всех элементов по очереди через соответствующее лямбда-выражение.
- Целевой метод делегата задается как лямбда-выражение

```
(student => student)
```

```
static Student GetStudent(Student s)
{
    |
    return s;
    |
}
```

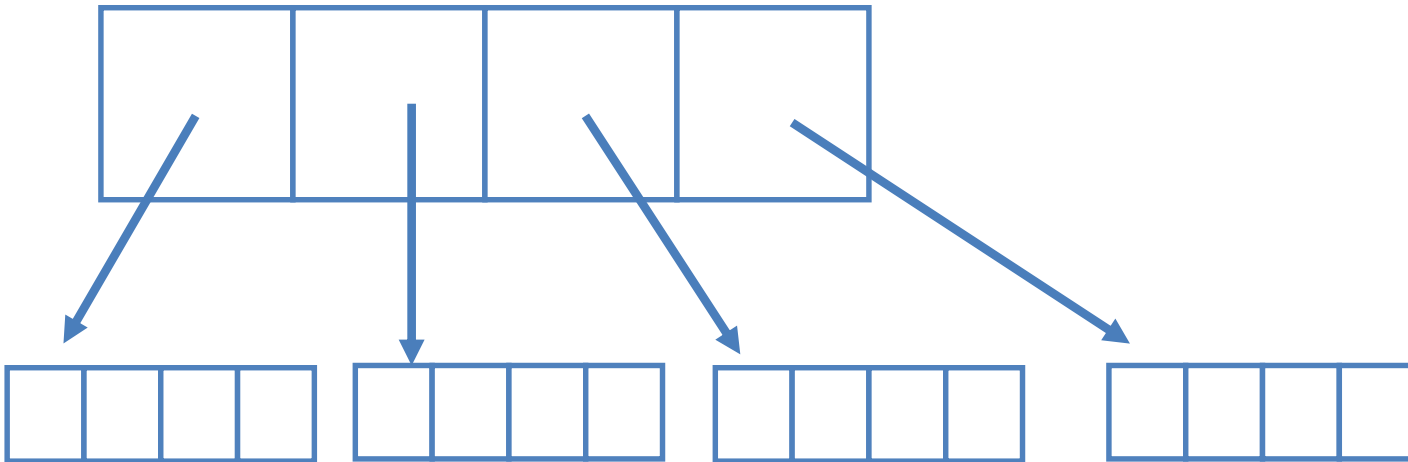
# Примеры запросов





# Коллекция коллекций

```
List<List<Person>> faculty = new List<List<Person>>();
```



# Примеры запросов: Фильтрация

```
IEnumerable<Person> res = from gr in collection
                           from item in gr
                           where item is Student && ((Student)item).Rating < 4
                           select item;
```

```
IEnumerable<Person> res=(collection.SelectMany(x => x)
    .Where(x => x is Student && ((Student)x).Rating < 4)
    .Select(x => x));
```

# Примеры запросов: Фильтрация

```
int res = (from gr in collection
           from s in gr
           where s is Student && ((Student)s).Rating < 4
           select s).Count();
```

---

```
int res = (collection.SelectMany(f => f)
           .Where(el => el is Student && ((Student)el).Rating < 4)
           .Select(x => x)).Count();
return res;
```

# Примеры запросов

- **Проекция**: позволяет спроектировать из текущего типа выборки какой-то другой тип.
- Для проекции используется оператор select.

```
var res = from el in c where el is Employee select  
new { Name = el.Name, Salary = ((Employee)  
el).Salary };
```

```
var res = c.Where(e=>e is Employee). Select(el =>  
new { Name = el.Name, Salary =  
((Employee)el).Salary });
```

# Примеры запросов

- Если требуется вернуть проекцию, как результат метода, то ее нужно преобразовать в массив с помощью метода `ToArray()`:

```
static Array NewTypeArray(ICollection<Person> c)
{
    var res = c.Where(e=>e is Employee). Select(el => new { Name =
    el.Name, Salary = ((Employee)el).Salary });
    return res.ToArray();
}
```

- Вызвать метод, который возвращает проекцию можно следующим образом:

```
Array arr=NewTypeArray(faculty1);
foreach (var x in arr)
    Console.WriteLine(x);
```

# Примеры запросов

- Дополнительные вычисления с помощью оператора let

var res = from el in list where el is Teacher

let **totalSalary** = ((Teacher)el).Salary +  
((Teacher)el).Salary \* ((Teacher)el).Bonus select  
new { Name = el.Name, TotalSalary = **totalSalary** };

Проекция

```
{ Name = Кириллов, TotalSalary = 75000 }
```

```
{ Name = Смирнова, TotalSalary = 85000 }
```

Для продолжения нажмите любую клавишу . . .

# Примеры запросов

- 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**.

# Примеры запросов

- **Сортировка**

```
var res = from el in list orderby el.Name select el;
```

```
var res = list.OrderBy (el => el.Name).Select(el=>el);
```

```
//по убыванию ключа
```

```
var res = from el in list orderby el.Name descending  
select el;
```

```
var res = list.OrderByDescending(el => el.Name).
```

```
Select(el=>el);
```

```
//множественные критерии
```

```
var res = from el in list orderby el.Name,el.Age select el;
```

```
var res = list.OrderBy(el =>
```

```
el.Name).ThenBy(el=>el.Age).Select(el=>el);
```



# 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.

# Примеры запросов

//Объединение

```
var res1 = (from el in faculty1 select el).Union(from el in  
faculty2 select el);
```

//разность

```
var res2 = (from el in faculty1 select el).Except(from el in  
faculty2 select el);
```

//пересечение

```
var res3 = (from el in faculty1 select el).Intersect( from el in  
faculty2 select el);
```

//сцепление и удаление дубликатов

```
var res4 = (from el in faculty1 select el).Concat(from el in  
faculty2 select el).Distinct();
```

# Агрегатные операции LINQ

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

//максимальное значение

```
var res = (from f in university from el in f where el is Employee
select ((Employee)el).Salary). Max();
```

```
var res = university.SelectMany(f => f).Select(e => e).Where(e
=> e is Student).Select(el => ((Employee)el).Salary). Max();
```

//среднее значение

```
var res = (from f in university from el in f where el is Employee
select ((Employee)el).Salary).Average();
```

```
var res = university.SelectMany(f => f).Select(e => e).Where(e
=> e is Student).Select(el => ((Employee)el).Salary).Average();
```

# Группировка

- Для группировки данных по определенным параметрам применяется оператор **group by** или метод **GroupBy()**.
- Результатом оператора group является выборка, которая состоит из групп.
- Каждая группа представляет объект `IGrouping<T1, T2>`: параметр `T1` указывает на тип ключа, а параметр `T2` - на тип сгруппированных объектов.
- Каждая группа имеет ключ, который мы можем получить через свойство `Key`: `g.Key`
- Все элементы группы можно получить с помощью дополнительной итерации.

# Примеры запросов

```
Console.WriteLine("Группировка по возрасту");  
var res = from el in faculty1 group el by el.Age;  
foreach (IGrouping<int, Person> g in res)  
{  
    Console.WriteLine(g.Key);  
    foreach (var e in g)  
        Console.WriteLine(e);  
}
```

# Примеры запросов

```
var grRes = from el in faculty1 group el by el.Age
into gr select new { Name = gr.Key, Count =
gr.Count() };
foreach (var g in grRes)
    Console.WriteLine($"{g.Name} - {g.Count}");
```

# Примеры запросов

- Аналогичные запросы можно построить с помощью метода расширения **GroupBy**:

```
var res = faculty1.GroupBy(el => el.Age);  
foreach (IGrouping<int, Person> g in res)  
{  
    Console.WriteLine(g.Key);  
    foreach (var e in g)  
        Console.WriteLine(e);  
}
```

# Примеры запросов

- Аналогичные запросы можно построить с помощью метода расширения **GroupBy**:

```
var grRes = faculty1.GroupBy(el =>  
el.Age).Select(gr => new { Name = gr.Key, Count  
= gr.Count() });  
foreach (var g in grRes)  
Console.WriteLine($"{g.Name} - {g.Count}");
```



# Методы All и Any

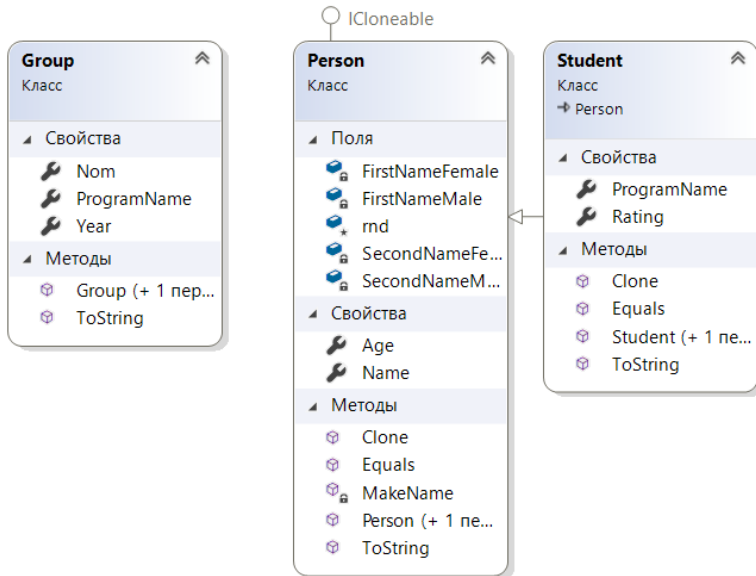
- Методы All, Any и Contains позволяют определить, соответствует ли коллекция определенному условию, и в зависимости от результата они возвращают true или false.
- Метод All проверяет, соответствуют ли все элементы условию.
- Метод Any позволяет узнать, соответствует ли хотя бы один элемент коллекции определенному условию.

# Примеры запросов

```
var res = (from f in faculty1 where f is Student  
&& ((Student)f).Rating < 4 select f).Any();
```

```
var res = (faculty1.Where(f => f is Student &&  
((Student)f).Rating > 4).Select(f => f)).All();
```

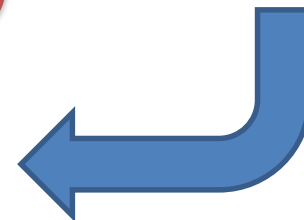
# Метод Join



Name	Age	Rating	Pr. Name
Иванов	21	6	РИС
Петров	22	2	РИС
Сидоров	23	8	БИ

Program Name	Year	Nom
РИС	23	1
БИ	22	2
ИАУП	21	1

Name	Age	Pr. Name	Year	Nom
Иванов	21	РИС	23	1
Петров	22	РИС	23	1
Сидоров	23	БИ	22	2



item2 in collection2 on  
item1.attribute  
item2.attribute  
}

collection1  
select new { attribute

Name	Age	Rating	Pr. Name
Иванов	21	6	РИС
Петров	22	2	РИС
Сидоров	23	8	БИ

collection2

attribute

Program Name	Year	Nom
РИС	23	1
БИ	22	2
ИАУП	21	1

Новый тип

# Метод Join

```
var result = from item1 in faculty
              from s1 in item1
              join t in groups on ((Student)s1).ProgramName equals t.ProgramName
              select new { Name = s1.Name, Program = t.ProgramName,
                           Group = t.ProgramName+"-"+t.Year+"-"+t.Nom };
foreach (var item in result)
{
    Console.WriteLine(item);
}
```

# Метод Join

```
var resultPI = from item1 in faculty
               from s1 in item1
               join t in groups on ((Student)s1).ProgramName equals t.ProgramName
               where t.ProgramName=="ПИ"
               select new
               {
                   Name = s1.Name,
                   Program = t.ProgramName,
                   Group = t.ProgramName + "-" + t.Year + "-" + t.Nom
               };
Console.WriteLine("Только ПИ");
foreach (var item in resultPI)
{
    Console.WriteLine(item);
}
```