

LINQ-запросы. Формирование запросов.

Доцент каф. АГУ Евдошенко О.И.

.NET Framework

.NET Framework - это независимая от языка среда программирования, разработанная корпорацией Microsoft. Кроме среды выполнения программы в Framework существуют библиотеки классов, которые упрощают разработку безопасных взаимодействующих с другими приложениями Windows и приложений на основе Интернет-технологий.



CLR (Common Language Runtime) – Среда Времени Выполнения или Виртуальная машина. Обеспечивает выполнение сборки. Основной компонент .NET Framework.

Запросы

Запрос представляет собой выражение, извлекающее данные из источника данных. Запросы обычно выражаются на специальном языке запросов.

Виды запросов:

- 1) SQL-запросы для реляционных баз данных
- 2) Xquery –запросы для XML – файлов.
- 3) Универсальный LINQ-запрос.

LINQ-запрос

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. Но вне зависимости от типа источника LINQ позволяет применить ко всем один и тот же подход для выборки данных.

LINQ упрощает ситуацию, предлагая единообразную модель для работы с данными в различных видах источников и форматов данных:

1. XML-документ
2. База данных SQL
3. Набор данных ADO.NET
4. Коллекция данных
5. Массив данных

Существует несколько разновидностей LINQ:

- **LINQ to Objects**: применяется для работы с массивами и коллекциями
- **LINQ to Entities**: используется при обращении к базам данных через технологию Entity Framework
- **LINQ to Sql**: технология доступа к данным в MS SQL Server
- **LINQ to XML**: применяется при работе с файлами XML
- **LINQ to DataSet**: применяется при работе с объектом `DataSet`
- **Parallel LINQ (PLINQ)**: используется для выполнения параллельных запросов

Операции запроса LINQ

1. Получение источника данных.

2. Создание запроса.

```
var query = from item in source
             where item <= значение
             select item;
```

3. Выполнение запроса.

```
string[] city = { "Москва", "Астрахань", "Саратов", "Ростов", "Астрахань", "Барселона" };
```

```
var selectedTeams = from t in city // определяем каждый объект из city как t
                    where t.ToUpper().StartsWith("Б") //фильтрация по критерию
                    orderby t // упорядочиваем по возрастанию
                    select t; // выбираем объект
```

Схема выполнения LINQ-запроса



Методы расширения LINQ

Кроме стандартного синтаксиса `from .. in .. select` для создания запроса LINQ мы можем применять специальные методы расширения, которые определены для интерфейса `IEnumerable`

Select: определяет проекцию выбранных значений

Where: определяет фильтр выборки

OrderBy: упорядочивает элементы по возрастанию

OrderByDescending: упорядочивает элементы по убыванию

ThenBy: задает дополнительные критерии для упорядочивания элементов возрастанию

ThenByDescending: задает дополнительные критерии для упорядочивания элементов по убыванию

Join: соединяет две коллекции по определенному признаку

GroupBy: группирует элементы по ключу

GroupJoin: выполняет одновременно соединение коллекций и группировку элементов по ключу

Reverse: располагает элементы в обратном порядке

All: определяет, все ли элементы коллекции удовлетворяют определенному условию

Any: определяет, удовлетворяет хотя бы один элемент коллекции определенному условию

Contains: определяет, содержит ли коллекция определенный элемент

Distinct: удаляет дублирующиеся элементы из коллекции

Удаление повторяющихся элементов и инвертирование

`var result = mas.Distinct();` - удаление дубликатов значений

`var result = mas.Reverse();` - изменение порядка элементов на противоположный

Переменные в запросах и оператор let

```
var people = from u in users
              let name = "Уважаемый " + u.Name
              select new
              {
                  Name = name,
                  Age = u.Age
              };
```


Операции Count, Sum

```
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet",  
"Chrysler", "Dodge", "BMW", "Ferrari", "Bentley", "Ford", "Lexus", "Mercedes", "Toyota",  
"Volvo", "Subaru", "Жигули"};
```

Операция **Count** возвращает количество элементов во входной последовательности. Эта операция имеет два прототипа, описанные ниже:

```
int count = cars.Count();           count = cars.Count(s => s.StartsWith("A"));
```

Операция **Sum** возвращает сумму числовых значений, содержащихся в элементах последовательности. Эта операция имеет два прототипа, описанные ниже:

```
IEnumerable<int> ints = Enumerable.Range(1, 100);  
int sum = ints.Sum();
```

```
long optionsSum = options.Sum(o => o > 5);
```

Операции Min, Max

```
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet",  
"Chrysler", "Dodge", "BMW", "Ferrari", "Bentley", "Ford", "Lexus", "Mercedes",  
"Toyota", "Volvo", "Subaru", "Жигули"};
```

```
int[] nums = { 34, 578, 200, 9, 72, 1399 };
```

Операция **Min** возвращает минимальное значение входной последовательности. Эта операция имеет четыре прототипа, которые описаны ниже:

```
int min = nums.Min(); //9
```

```
string auto = cars.Min(); // Alfa Romeo
```



Элемент с минимальным
значением в алфавитном
порядке

```
min = cars.Min(a => a.Length>5);
```

Операция **Max** возвращает максимальное значение из входной последовательности. Эта операция имеет четыре прототипа, полностью идентичных прототипам операции Min

```
string auto = cars.Max(); //???
```

Операции Average

Операция **Average** возвращает среднее арифметическое числовых значений элементов входной последовательности. Эта операция имеет два прототипа, описанные ниже:

```
IEnumerable<int> nums = Enumerable.Range(1, 10);
```

```
double average = nums.Average();
```

```
double Average = nums.Average(o => o > 5);
```

Фильтрация

Фильтрация последовательности на основе выбранного предиката

```
int[] mas = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
  
    var result = mas.Where(n => n % 2==0);  
// 2 4 6 8 10
```

```
int[] scores = { 90, 71, 82, 93, 75, 82 };  
int highScoreCount = scores.Where(n => n > 80).Count();
```

Возвращает цепочку элементов последовательности, удовлетворяющую условию

```
int[] mas = { 2, 2, 5, 4, 5, 6, 7, 8, 9, 10 };  
  
    var result = mas.TakeWhile(n => n % 2==0);  
// 2 2  
  
var result = mas.TakeWhile(n => n < 6);
```

Фильтрация

Пропускает элементы пока они удовлетворяют заданному условию

```
int[] mas = { 2, 2, 5, 4, 5, 6, 7, 8, 9, 10 };
```

```
    var result = mas.SkipWhile(n => n % 2==0);  
//5 4 5 6 7 8 9 10
```

Возвращает указанное число идущих подряд элементов

```
int[] mas = { 2, 2, 5, 4, 5, 6, 6, 8, 9, 10 };
```

```
    var result = mas.Take(5);  
//2 2 5 4 5
```

Пропускает указанное число элементов последовательности и возвращает остальные:

```
IEnumerable<int> result = mas.Skip(4);
```

```
//5 6 6 8 9 10
```

Сортировка

```
int[] mas = { 6, 2, 5, 4, 9, 6, 10, 8, 5, 2 };
```

```
var result = mas.OrderBy(n => n);
```

```
//2 2 4 5 5 6 6 8 9 10
```

```
var result = mas.OrderByDescending(n => n);
```

```
//10 9 8 6 6 5 5 4 2 2
```

```
var result = from объект in Источник  
              orderby Поле1, Поле2  
              select объект;
```

```
var result = Источник.OrderBy(u => u.Поле1).ThenBy(u => u.Поле2);
```

Операция StartsWith, EndsWith

```
string[] cars = { "Nissan", "Aston Martin", "Chevrolet", "Alfa Romeo",  
"Chrysler", "Dodge", "BMW", "Ferrari", "Audi", "Bentley", "Ford", "Lexus",  
"Mercedes", "Toyota", "Volvo", "Subaru", "Жигули"};
```

```
var sequence = cars.Where(p => p.StartsWith("F"));
```

```
foreach (string s in sequence)  
    Console.WriteLine(s);
```



A screenshot of a console window with a black background and white text. The text displays the results of the LINQ query: "Ferrari" on the first line and "Ford" on the second line. The window has a standard Windows-style title bar and scrollbars on the right side.

```
string auto = cars.Where(p => p.EndsWith("s"));
```

```
IEnumerable<string> sequence = cars.Where((p, i) => (i % 2) == 1);
```

Операции First, Last, Single

```
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet",  
"Chrysler", "Dodge", "BMW", "Ferrari", "Bentley", "Ford", "Lexus", "Mercedes",  
"Toyota", "Volvo", "Subaru", "Жигули"};
```

```
string auto = cars.First(); // Alfa Romeo
```

```
string auto = cars.First(p => p.StartsWith("C")); //??? Chevrolet
```

```
string auto = cars.Last(); // Жигули
```

```
string auto = cars.Last(p => p.StartsWith("C")); //??? Chrysler
```

```
var emp = Студенты.Where(n => n.Фамилия == "Иванов").Single();
```

Возвращает единственный элемент последовательности или единственный элемент последовательности, соответствующий предикату

```
var emp = Студенты.Single(n => n.Фамилия == "Иванов");
```

Ошибка, если
последовательность
содержит более одного
соответствующего
элемента

Операции First..., Last..., Single... OrDefault

```
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet", "Chrysler", "Dodge", "BMW",  
                  "Ferrari", "Bentley", "Ford", "Lexus", "Mercedes", "Toyota", "Volvo", "Subaru", "Жигули"};
```

```
string auto = cars.Take(0).FirstOrDefault();  
Console.WriteLine(auto == null ? "\n\tНичего не найдено" : auto);
```

```
string auto = cars.Take(0).LastOrDefault();
```

```
string auto = cars.LastOrDefault(p => p.StartsWith("A"));
```

```
Console.WriteLine(auto == null ? "Ничего не найдено" : auto);
```

```
string firstLongName = cars.FirstOrDefault(name => name.Length > 5);
```

Возвращает первый (последний, единственный) элемент, найденный во входной последовательности. Если последовательность пуста, возвращается default(T). Для ссылочных и допускающих null типов значением по умолчанию является null.

Операции DefaultIfEmpty

Операция **DefaultIfEmpty** возвращает последовательность, содержащую элемент по умолчанию, если входная последовательность пуста.

```
try
{
    string porsche = cars.Where(n =>n.Equals("Porsche")).First();

    if (porsche != null)
        MessageBox.Show("Автомобиль Porsche найден");
    else
        MessageBox.Show("Автомобиль Porsche не найден");
}

catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}

string porsche = cars
    .Where(n => n.Equals("Porsche"))
    .DefaultIfEmpty("Не найдено :(")
    .First();
```

Задача

```
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet",  
"Chrysler", "Dodge", "BMW", "Ferrari", "Bentley", "Ford", "Lexus", "Mercedes",  
"Toyota", "Volvo", "Subaru", "Жигули"};
```

```
IEnumerable<string> auto = cars.Take(5).Concat(cars.Skip(5));
```

```
foreach (string str in auto)  
    Console.WriteLine(str);
```

Дано целое число K (>0) и строковая последовательность A . Строки последовательности содержат только цифры и заглавные буквы латинского алфавита. Извлечь из A все строки длины K , оканчивающиеся цифрой.

```
int K = 4;  
string[] mas = { "AXS1", "2TER", "B2GF", "LK3IK1" };  
  
var result = mas.Where  
(n => n.Length == K &&  
int.TryParse(n.Substring(n.Length - 1, 1), out x));
```

Операции Any, All

Операция **Any** возвращает true, если любой из элементов входной последовательности отвечает условию. Эта операция имеет два прототипа, которые описаны ниже:

```
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet",  
"Chrysler", "Dodge", "BMW", "Ferrari", "Bentley", "Ford", "Lexus", "Mercedes",  
"Toyota", "Volvo", "Subaru", "Жигули"};
```

```
bool anyNull = Enumerable.Empty<string>().Any(); //False (не содержит элементы)
```

```
bool anyCars = cars.Any(); //True (содержит элементы)
```

```
anyCars = cars.Any(s => s.StartsWith("X")); // ??? False
```

Операция **All** возвращает true, если каждый элемент входной последовательности отвечает условию. Операция All имеет один прототип, описанный ниже:

```
bool all = cars.All(s => s.StartsWith("A"));
```

Правда ли, что все элементы коллекции Cars начинаются с A: False

```
all = cars.All(s => s.Length > 2);
```

Правда ли, что все элементы коллекции Cars длинее 2х символов: ????

Операции Aggregate

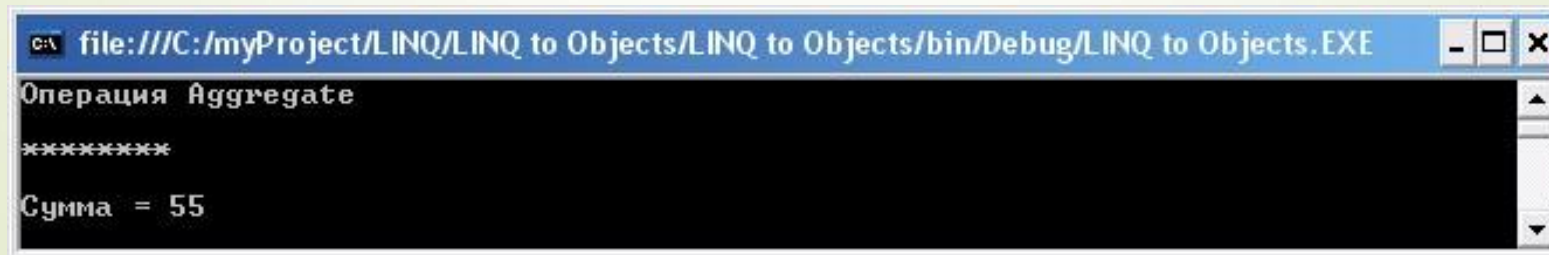
Операция **Aggregate** выполняет указанную пользователем функцию на каждом элементе входной последовательности, передавая значение, возвращенное этой функцией для предыдущего элемента, и возвращая ее значение для последнего элемента. Эта операция имеет два прототипа, которые описаны ниже:

```
int N = 5;
```

```
int agg = Enumerable  
    .Range(1, N)  
    .Aggregate((av, e) => av * e);  
Console.WriteLine("{0}! = {1}", N, agg); //5! = 120
```

```
int N = 10;
```

```
int agg = Enumerable  
    .Range(1, N)  
    .Aggregate(0, (s, i) => s + i);
```



```
C:\> file:///C:/myProject/LINQ/LINQ to Objects/LINQ to Objects/bin/Debug/LINQ to Objects.EXE  
Операция Aggregate  
*****  
Сумма = 55
```

Операции Select

```
string[] cars = { "Nissan", "Aston Martin", "Chevrolet", "Alfa Romeo", "Chrysler",  
"Dodge", "BMW", "Ferrari", "Audi", "Bentley", "Ford", "Lexus", "Mercedes", "Toyota",  
"Volvo", "Subaru", "Жигули"};
```

```
IEnumerable<int> sequence = cars.Select(p => p.Length);
```

```
var sequence = cars.Select(p => new { p, p.Length });
```

```
var carObj = cars.Select(p => new { LastName = p, Length = p.Length });
```

Оператор **Select** создает объект анонимного типа. Результат будет содержать набор объектов данного анонимного типа, в котором определены два свойства: `LastName` и `Length`.

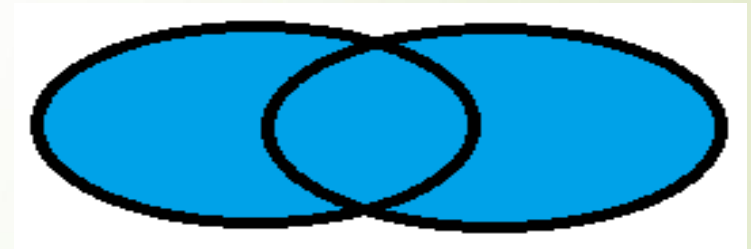
```
foreach (var i in carObj)  
Console.WriteLine("Автомобиль {0} имеет длину {1} символов", i.LastName, i.Length);
```

```
var carObj = cars.Select((p, i) => new { Index = i + 1, LastName = p });
```

Union

```
public static IEnumerable<T> Union<T>(
    this IEnumerable<T> first,
    IEnumerable<T> second);
```

Объединением двух последовательностей называется последовательность, которая включает в себя все записи исходных последовательностей без повторов.



Объединение двух последовательностей:

```
IEnumerable<string> concat = first.Concat<string>(second);
```

Объединение двух последовательностей без повторений:

```
IEnumerable<string> union = first.Union<string>(second);
```

```
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet",  
"Chrysler", "Dodge", "BMW", "Ferrari", "Bentley", "Ford", "Lexus", "Mercedes", "Toyota",  
"Volvo", "Subaru", "Жигули"};
```

```
IEnumerable<string> first = cars.Take(5);
```

```
IEnumerable<string> second = cars.Skip(4);
```

```
IEnumerable<string> concat = first.Concat<string>(second);
```

```
IEnumerable<string> union = first.Union<string>(second);
```

```
Console.WriteLine(@"Машины: {0} элементов
```

```
first: {1} элементов
```

```
second: {2} элементов
```

```
concat: {3} элементов
```

```
union: {4} элементов",
```

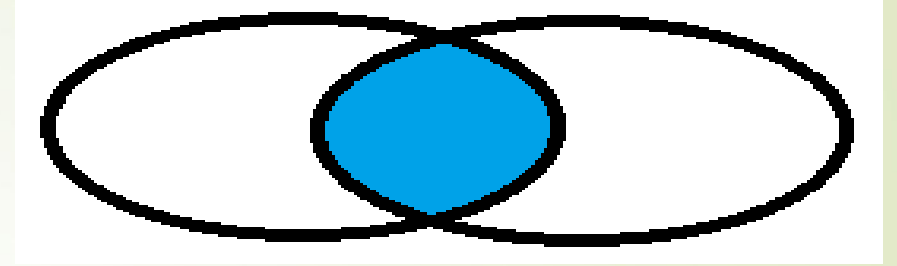
```
cars.Count(), first.Count(), second.Count(), concat.Count(), union.Count());
```



```
file:///C:/myProject/LINQ/LINQ to Objects/bin/Debug/LINQ to Objects.EXE  
Cars: 17 элементов  
first: 5 элементов  
second: 13 элементов  
concat: 18 элементов  
union: 17 элементов
```


Intersect

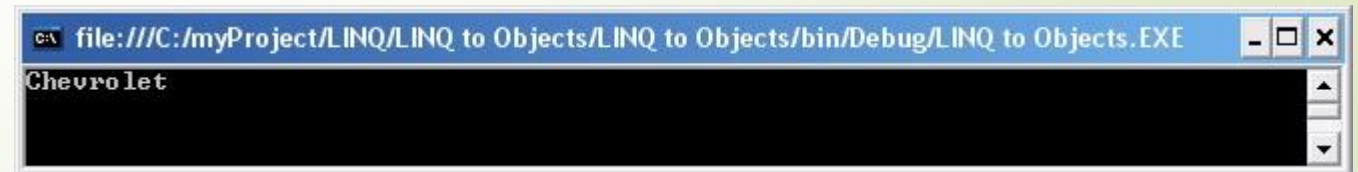
```
public static IEnumerable<T> Intersect<T>(
    this IEnumerable<T> first,
    IEnumerable<T> second);
```



Пересечение двух последовательностей есть подмножество записей, принадлежащих обоим последовательностям.

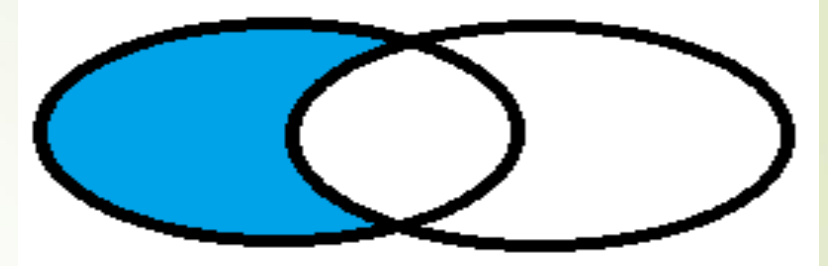
```
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi",
    "Nissan", "Chevrolet", "Chrysler", "Dodge", "BMW", "Ferrari",
    "Bentley", "Ford", "Lexus", "Mercedes", "Toyota", "Volvo",
    "Subaru", "Жигули"};
IEnumerable<string> first = cars.Take(5);
IEnumerable<string> second = cars.Skip(4);
IEnumerable<string> auto = first.Intersect(second);
```

```
foreach (string s in auto)
    Console.WriteLine(s);
```



Except

```
public static IEnumerable<T> Except<T>(
    this IEnumerable<T> first,
    IEnumerable<T> second)
```



Разность возвращает последовательность, содержащую все записи, которые принадлежат первой из заданных последовательностей и не принадлежат второй

```
int[] arr1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int[] arr2 = { 6, 7, 8, 9 };
```

```
IEnumerable<int> nums = arr1.Except<int>(arr2);
```

```
foreach (int i in nums)
    Console.WriteLine(i);
```



Задание

В центре детского творчества имеется два кружка: драматический и математический. В базе хранится информацию о фамилиях участников каждого кружка. Вывести участников, которые занимаются в обоих кружках и фамилия начинается на букву «А».

Делегаты и анонимные методы в запросах LINQ

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>**

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7 };
var result = numbers.Where(MoreThanTen);

private static bool MoreThanTen(int i)
{
    return i > 10;
}

int[] numbers = { -2, -1, 0, 1, 2, 3, 4, 5, 6, 7 };
var result = numbers.Where(i => i > 0).Select(Factorial);

static int Factorial(int x)
{
    int result = 1;
    for (int i = 1; i <= x; i++)
        result *= i;
    return result;
}
```

Работа с датами и временем

Установка времени:

`DateTime date1 = new DateTime(2019, 7, 20, 18, 30, 25);` // год - месяц - день - час - минута - секунда (20.07.2019 18:30:25)

Для добавления дат используется ряд методов:

Add(DateTime date): добавляет дату date

AddDays(double value): добавляет к текущей дате несколько дней

AddHours(double value): добавляет к текущей дате несколько часов

AddMinutes(double value): добавляет к текущей дате несколько минут

AddMonths(int value): добавляет к текущей дате несколько месяцев

AddYears(int value): добавляет к текущей дате несколько лет

```
Console.WriteLine(date1.ToLocalTime()); // 20.07.2019 21:30:25
```

```
Console.WriteLine(date1.ToUniversalTime()); // 20.07.2019 15:30:25
```

```
Console.WriteLine(date1.ToLongDateString()); // 20 июля 2019 г.
```

```
Console.WriteLine(date1.ToShortDateString()); // 20.07.2019
```

```
Console.WriteLine(date1.ToLongTimeString()); // 18:30:25
```

```
Console.WriteLine(date1.ToShortTimeString()); // 18:30
```

```
Console.WriteLine(date1.AddHours(3));
```

```
Console.WriteLine(date1.AddHours(-3));
```

Кортежи

Кортежи предоставляют удобный способ для работы с набором значений

```
PM> Install-Package "System.ValueTuple"
```

```
var tuple = (5,10,4); // неявное определение типа
Console.WriteLine(tuple.Item1); // 5
```

```
(int, int) tuple = (5, 10);
```

```
(string, int, double) person = ("Елизавета II", 25, 81.23);
```

Названия полям кортежа:

```
var tuple = (count: 5, sum: 10);
Console.WriteLine(tuple.count); // 5
Console.WriteLine(tuple.sum); // 10
```

```
var (name, age) = ("Елизавета II", 23);
Console.WriteLine(name); // Елизавета II
Console.WriteLine(age); // 23
```

Кортежи

```
var tuple = GetNamedValues(new int[] { 1, 2, 3, 4, 5, 6, 7 });
Console.WriteLine(tuple.count);
Console.WriteLine(tuple.sum);

private static (int sum, int count) GetNamedValues(int[] numbers)
{
    var result = (sum: 0, count: 0);
    for (int i = 0; i < numbers.Length; i++)
    {
        result.sum += numbers[i];
        result.count++;
    }
    return result;
}

var (name, age) = GetTuple(("Елизавета II", 23), 12);
Console.WriteLine(name);    // Елизавета II
Console.WriteLine(age);     // 35

private static (string name, int age) GetTuple((string n, int a) tuple, int x)
{
    var result = (name: tuple.n, age: tuple.a + x);
    return result;
}
```

Интерполяция строк

```
int a = 1;  
int b = 2;
```

```
Console.WriteLine($"A= {a} B= {b}");
```

Знак доллара перед строкой указывает, что будет осуществляться интерполяция строк. Внутри строки опять же используются плейсхолдеры {...}, только внутри фигурных скобок уже можно напрямую писать те выражения, которые мы хотим вывести.

```
string result = $"{a} + {b} = {a + b}";  
Console.WriteLine(result); // 1 + 2 = 3
```

Настраиваемые форматы

```
long number = 19876543210;  
Console.WriteLine($"{number:+# ### ##}"); // +1 987 654 32 10
```

Добавление пробелов до и после

```
Console.WriteLine($"A: {a,-5} B: {b}"); // пробелы после  
Console.WriteLine($"A: {a,5} B: {b}"); // пробелы до
```