13. LINQ (Language-Integrated Query)

Доц.Виолета Божикова

Основи на LINQ

- LINQ (Language-Integrated Query) представлява прост и удобен език за заявки към източник на данни.
- В качеството на източник на данни може да е обект, реализиращ интерфейс **IEnumerable** (например: **стандартни колекция**, **масив**, **набор от данни DataSet**, **документ XML**).
- Независимо от източника на данни, LINQ позволява да се използва **един и същ подход за извличане на извадка**.
- Съществуват няколко разновидности на LINQ:
  - **LINQ to Objects**: използва се за работа с масиви и колекции
  - **LINQ to Entities**: използва се при обращение към БД чрез технология Entity Framework
  - **LINQ to DataSet**: използва се за работа с обекти от типDataSet
  - **LINQ to XML**: използва се за работа с файлове XML
  - **Parallel LINQ (PLINQ)**: използва се за изпълнение на паралелни заявки

!!!Лекцията се концентрира върху **LINQ to Objects**.

В какво е удобството на LINQ?

Един прост пример: Да намерим от даден масив стринговете, започващи с дадена буква и да сортираме получената извадка.

Реализация без LINQ:

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

var selectedTeams = new List<string>();
foreach(string s in teams)
{
   if (s.ToUpper().StartsWith("Б"))
      selectedTeams.Add(s);
}
selectedTeams.Sort();
foreach (string s in selectedTeams)   Console.WriteLine(s);
```

Сега с LINQ:

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

var selectedTeams = from t in teams // t е всеки обект от teams
                    where t.ToUpper().StartsWith("Б") //филтрация по критерия
                    orderby t  // нареждаме възходящо извадката
                    select t; // връща се извадка, в качеството на резултат от LINQ

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

- За да използваме функционалност LINQ, трябва да включим пространството от имена **System.LINQ**.
- И така, с използване на LINQ, кодът стана по-къс и прост. Целият израз се запише по-просто в един ред:

```
var selectedTeams =
 from t in teams // t е всеки обект от teams
 where t.ToUpper().StartsWith("Б")
                        //филтрация по критерия
 orderby t          // нареждаме възходящо извадката
 select t;
     // връща се извадка, в качеството на резултат от LINQ
```

- Най-простото определение на заявка LINQ изглежда по следния начин:

```
from променлива in  набор_обекти
select променлива;
```

И така, какво прави LINQ?

- Изразът **from t in teams** преглежда всички елементи на масива **teams** и определя всеки елемент като **t**. Използвайки променлива **t** ние можем да извършим над нея различни операции.
- Независимо, че не указваме типа на **t**, изразът LINQ се явява строго типизиран, тоест средата автоматически распознава, че набор **teams** се състои от обекти **string**, ето защо и променливата **t** ще бъде разглеждана като стринг.
- След това, с помощта на оператор **where** се прави филтрация на обектите, и ако обектът **t** съответства на критерия (в дадения случай началната буква да е "Б"), то този обект **t** се предава по-нататък тоест попада в избраните обекти.
- Оператор **orderby** подрежда по нарастване **избраните обекти (извадка**), тоест сортира избраните обекти във възходящ ред.
- Оператор **select** предава селектираните стойности на резултатната извадка, която се връща в качеството на резултат от LINQ-израза.
- В дадения случай, резултатът от LINQ израза се явява обект от тип IEnumerable<T>. Нерядко резултатната извадка се определя с помощта на ключовата дума var, тогава компилаторът на етапа на компиляция сам извежда типа.
- Преимущество на подобни заявки се явява и това, че те интуитивно приличат на SQL заявки, независимо че имат някои различия.

Методи за разширяване на LINQ

- Освен стандартния синтаксис **from .. in .. select** за създаване на заявка **LINQ**, ние може да приложим и специални методи за разширение, които са дефинирани в интерфейса **IEnumerable**.
- В този смисъл, методите на **LINQ** - **Where** и **orderby** реализират същата функционалност:

Например:

- ...

```
string[] teams = { "Бавария", "Борусия", "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);
```

Методи за разширяване на LINQ

- Заявката **teams.Where(t=>t.ToUpper().StartsWith("Б")).OrderBy(t => t)** ще бъде аналогична на предната.
- Тя се състои от веригата методи **Where** и **OrderBy**. В качество на аргументи, тези методи приемат **делегат** или **ламбда - израз**.
- Не всеки метод за разширение има аналог сред операторите на LINQ, но в този случай може да се съчетаят двата подхода.
- Например, използвайки стандартният синтаксис на **LINQ** и метода за разширение Count(), връщаме количеството на елементите в извадката:

Методи за разширяване на LINQ

- Заявката:

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

ще бъде аналогична на предната.

- Тя се състои от веригата методи **Where** и **OrderBy**. В качество на аргументи, тези методи приемат **делегат** или **ламбда - израз**.
- Не всеки метод за разширение има аналог сред операторите на LINQ, но в този случай може да се съчетаят двата подхода.
- Например, използвайки стандартният синтаксис на **LINQ** и метода за разширение Count(), връщаме количеството на елементите в извадката:

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

Списък на използваните методи за разширение на LINQ

Select: определя проекция на избраните стойности

Where: определят филтър за избор

OrderBy: подрежда елементите по нарастване

OrderByDescending: подрежда елементите по намаляване

ThenBy: задава допълнителни критерии за подреждане на елементите по нарастване

ThenByDescending: задава допълнителни критерии за подреждане на елементите по намаляване

Join: съединява 2 колекции по определен признак

GroupBy: групира елементите по ключ

ToLookup: групира елементите по ключ, при това всички елементи се добавят в речника

GroupJoin: изпълнява едновременно съединение на колекция и групиране на елементите по ключ
Reverse: разполага елементите в обратен ред
All: определя, дали всички елементи на колекцията удовлятворяват определено условие
Any: определя, дали поне един елемент на колекцията удовлятворява определено условие
Contains: определя, съдержа ли колекцията определен елемент
Distinct: премахва дублиращите се елементи от коллекцията
Except: връща разликата на 2 колекции, тоест тези елементи, които се съдържат само в една колекция
Union: обединява две еднородни колекции
Intersect: връща сечението на 2 колекции, тоест тези елементи, които се срещат в 2-те колекции
Count: преброява количеството на елементите в колекцията, които удовлетворяват определено условие

Sum: изчислява сумата на числовите стойности в колекцията
Average: изчислява средната стойност на числовите стойности на колекцията
Min: намира минималната значение
Max: намира максималната значение
Take: избира определено количество елементи
Skip: пропуска определено количество елементи
TakeWhile: връща верига от елементи на последователността докато, условието е истинно
SkipWhile: пропуска елементи в последователността докато те удовлетворяват зададено условие, и след това връща оставащите елементи
Concat: обединява две колекции
Zip: обединява две колекции в соответствие с определено условие
First: избира първия елемент на колекцията

FirstOrDefault: избира първия елемент на колекцията или връща стойност по подразбиране

Single: избира единствен елемент на колекцията, ако колекцията съдържа повече или по-малко от един елемент, то се генерира изключение

SingleOrDefault: избира първия елемент на колекцията или връща стойност по подразбиране

ElementAt: избира елемент от последователността по определен индекс

ElementAtOrDefault: избира първия елемент на колекцията или връща стойност по подразбиране

Last: избира последния елемент на колекцията

LastOrDefault: избира последния елемент на колекцията или връща стойност по подразбиране

Филтрация на извадката и проекция

Филтрация на извадката

- За избор на елементи от някакъв набор елементи се използва метод Where. Например, да изберем всички четни елементи, по-големи от 10 от масив.
- Филтрация с помощта на операторите на LINQ:

```
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);
```

- Тук се използва конструкция from: **from i in numbers**
- Същата заявка, с помощта на метод на разширение:

```
int[] numbers = { 1, 2, 3, 4, 10, 34, 55, 66, 77, 88 };
IEnumerable<int> evens = numbers.Where(i => i % 2 == 0 && i > 10);
```

- Ако изразът в метод **Where** за определен елемент е true (в дадения случай, ако изразът **i % 2 == 0 && i > 10** е true), то даденият елемент попада в резултантната извадка

Извадка от сложни обекти

- Да допуснем, че имаме потребителски клас:

```
class User
{
    public string Name { get;set; }
    public int Age { get; set; }
    public List<string> Languages { get; set; }
    public User()
    {
        Languages = new List<string>();
    }
}
```

Да създадем набор от потребители (н.р списък **users** от **User**) и да изберем тези, които са по-големи от 25 години:

```
List<User> users = new List<User>
{
    new User {Name="Том", Age=23, Languages = new List<string> {"английски", "немски" }},
    new User {Name="Боб", Age=27, Languages = new List<string> {"английски", "френски" }},
    new User {Name="Джон", Age=29, Languages = new List<string> {"английски", "испански" }},
    new User {Name="Элис", Age=24, Languages = new List<string> {"испански", "немски" }}
};
```

```
var selectedUsers = from user in users
                    where user.Age > 25
                    select user;
foreach (User user in selectedUsers)
Console.WriteLine("{0} - {1}", user.Name,
user.Age);
```

Конзолен изход:

Боб - 27

Джон - 29

- Аналогичната заявка с помощта на метода за разширение Where:

```
var selectedUsers = users.Where(u=> u.Age > 25);
```

Сложни филтри

- Например, в клас user има списък на езиците, които потребителят владее. Ако желаем да филтрираме ползвателите освен по възраст и по някакъв език, например английски, то имаме:

```
var selectedUsers = from user in users
                    from lang in user.Languages
                    where user.Age < 28
                    where lang == "английски"
                    select user;
```

- Резултат:

Том - 23

Боб - 27

- За създаване на аналогично запитване, с помощта на методите за разширение тогава трябва да използваме метод SelectMany:

```
var selectedUsers = users.SelectMany(u => u.Languages,
                                     (u, l) => new { User = u, Lang = l })
    .Where(u => u.Lang == "английски" && u.User.Age < 28)
    .Select(u=>u.User);
```

- Метод SelectMany() приема последователност в качество на първи параметър (това е тази последователност, която трябва да се проектира), а в качество на втори параметър – функция за преобразуване, която се прилага за всеки елемент.
- На изхода SelectMany() връща 8 двойки "потребител - език" **(new { User = u, Lang = l })**, към които по нататък се прилага филтъра с помощта на Where.

Проекция

- Проекцията позволява да се проектира от текущия тип извадка някакъв друг тип.
- За проекция се използва оператор select.
- Да допуснем, че имаме набор обекти от следния клас, представящ потребителя:

```
class User
{  public string Name { get;set; }
    public int Age { get; set; }
}
```

Нека ни е нужен не целия обект, а само неговото свойство Name:

```
List<User> users = new List<User>();
users.Add(new User { Name = "Sam", Age = 43 });
users.Add(new User { Name = "Tom", Age = 33 });
/*нужен ни е не целия обект, а само неговото свойство Name:*/
var names = from u in users select u.Name;
foreach (string n in names)
    Console.WriteLine(n);
```

- Резултатът от LINQ израза ще бъде набор от стрингове, според израза: select u.Name
- Т.е избират се само стойностите на свойство Name.

Аналогично може да се създадат обекти от друг тип, в това число анонимен:

```
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
        { /* Тук оператор select създава обект от анонимен тип, използвайки текущия обект User. */
            FirstName = u.Name,
            DateOfBirth = DateTime.Now.Year - u.Age
        };
/* резултатът ще съдържа набор от обекти от дадения анонимен тип, в който са дефинирани 2 свойства: FirstName и DateOfBirth */
 foreach (var n in items)
 Console.WriteLine("{0} - {1}", n.FirstName, n.DateOfBirth);
```

- Тук оператор select създава обект от анонимен тип, използвайки текущия обект User. И сега резултатът ще съдържа набор от обекти от дадения анонимен тип, в който са дефинирани 2 свойства: FirstName и DateOfBirth.
- В качеството на алтернатива бихме могли да използваме метода за разширение Select():

```
// набор от имена
var names = users.Select(u => u.Name);

// набор от обекти от анонимен типа
var items = users.Select(u => new
{
   FirstName = u.Name,
   DateOfBirth = DateTime.Now.Year - u.Age
}
);
```

Променливи в заявки и оператор let

- Понякога възниква необходимост за извършване на допълнителни изчисления в заявки LINQ. За тези цели може да се зададат в заявките собствени променливи с помощта на оператора let:

```
List<User> users = new List<User>()
{  new User { Name = "Sam", Age = 43 },
   new User { Name = "Tom", Age = 33 }
};
 var people = from u in users
          let name = "Mr. " + u.Name /*създава променлива name, значението на която е равно на "Mr. " + u.Name*/
          select new
          {  Name = name,   Age = u.Age
          };
```

Набор от няколко източника

- В LINQ може да се избират обекти не само от един, но от по-голямо количество източници:
- Например, да вземем класовете Phone и User:

```
class Phone
{
    public string Name { get; set; }
    public string Company { get; set; }
}
class User
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

- Да създадем 2 различни източника данни users и phones. Да направим избора:

```
List<User> users = new List<User>()
{
    new User { Name = "Sam", Age = 43 },
    new User { Name = "Tom", Age = 33 }
};
List<Phone> phones = new List<Phone>()
{
    new Phone {Name="Lumia 630",
Company="Microsoft" },
    new Phone {Name="iPhone 6",
Company="Apple"},
};
```

- Да направим избора:

```
var people = from user in users
             from phone in phones
 select new { Name = user.Name, Phone = phone.Name };

foreach (var p in people)
    Console.WriteLine("{0} - {1}", p.Name, p.Phone);
```

- Конзолен изход:

Sam - Lumia 630

Sam - iPhone 6

Tom - Lumia 630

Tom - iPhone 6

- По този начин при избор от 2 източника всеки елемент на първия източник ще се съпоставя с всеки елемент на втория, тоест получават се 4 двойки.

Сортировка

- За сортировка на набора данни по нарастващ ред се използва оператор orderby (сортировка по нарастване):

```
int[] numbers = { 3, 12, 4, 10, 34, 20, 55, -66, 77, 88, 4 };
var orderedNumbers = from i in numbers
                     orderby i //orderby i descending
                     select i;
foreach (int i in orderedNumbers)
   Console.WriteLine(i);
```

- Да разгледаме последно примера. Да допуснем, че трябва да се сортира извадка от сложни обекти. Тогава в качество на критерий можем да укажем свойство на класа на обекта:

```
List<User> users = new List<User>()
{
   new User { Name = "Tom", Age = 33 },
   new User { Name = "Bob", Age = 30 },
   new User { Name = "Tom", Age = 21 },
   new User { Name = "Sam", Age = 43 }
};

var sortedUsers = from u in users
                  orderby u.Name // descending
                  select u;

foreach (User u in sortedUsers)
   Console.WriteLine(u.Name);
```

- Вместо оператора orderby може да се използва метод за разширение 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 = 33 },
    new User { Name = "Bob", Age = 30 },
    new User { Name = "Tom", Age = 21 },
    new User { Name = "Sam", Age = 43 }
};
var sortedUsers = users.OrderBy(u=>u.Name);
```

- В намаляващ ред:

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

Множествени критерии за сортировка

- В извадките от сложни обекти понякога възниква ситуация, когато трябва да се сортира не един, а по няколко полета едновременно.
- Затова в заявката LINQ всички критерии се указват в ред по приоритет, разделени със запетая:

```
List<User> users = new List<User>()
{
    new User { Name = "Tom", Age = 33 },
    new User { Name = "Bob", Age = 30 },
    new User { Name = "Tom", Age = 21 },
    new User { Name = "Sam", Age = 43 }
};
var result = from user in users
             orderby user.Name, user.Age, user.Name.Length
             select user;
foreach (User u in result)
    Console.WriteLine("{0} - {1}", u.Name, u.Age);
```

- Резултат от програмата:

```
Alice - 28
Bob - 30
Sam - 43
Tom - 21
Tom - 33
```

- С помощта на методите за разширение това същото може да направим и чрез метод ThenBy()(за сортировка във втзходящ ред) и ThenByDescending() (за сортировка по низходящ ред):

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

- Резултатът ще бъде аналогичен на предния.

Работа с множества. Разлика на множества

- Освен методи върху извадка, LINQ има и няколко метода за работа с множества: разлика, обединение и сечение.
- С помощта на метода Except може да се получи разликата между 2 множества:

```
string[] soft = { "Microsoft", "Google", "Apple"};
string[] hard = { "Apple", "IBM", "Samsung"};

// разлика на множества soft и hard
var result = soft.Except(hard);

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

- В дадения случай от масива soft се извеждат всички елементи, които не са в масива hard:

Microsoft

Google

Сечение на множества

- За извеждане на общите елементи за 2 масива се използва метод Intersect:

```
string[] soft = { "Microsoft", "Google", "Apple"};
string[] hard = { "Apple", "IBM", "Samsung"};

// сечение на множества soft и hard
var result = soft.Intersect(hard);

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

- В дадения случай се извеждат всички елементи, които са общи за 2-та масива:

Apple

Обединение на множества

- За обединение на 2 множества (масиви) се използва метод Union. Неговият резултат се явява ново множество, в което има елементи, както от едното, така и от второто множество. Повтарящите се елементи се добавят в резултата само веднъж:

```
string[] soft = { "Microsoft", "Google", "Apple"};
string[] hard = { "Apple", "IBM", "Samsung"};

// обединение на множества
var result = soft.Union(hard);
// var result = soft.Concat(hard); - прави същото
 foreach (string s in result)
    Console.WriteLine(s);
```

- В дадения случай се извеждат всички елементи, които са общи за 2-та масива:

```
Microsoft
Google
Apple
IBM
Samsung
```

Премахване на дубликати

- За премахване на дъбликати в множеството се използва метод Distinct:

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

- Последователното прилагане на методите Concat и Distinct ще бъде подобно на действието на метода Union.

Агрегатни операции

- Към агрегатните операции се отнасят различни операции над извадка, например, получаване на броя на елементите, получаване на минималната, максималната и средната стойност в извадка, а така също и сума на стойностите.

Метод Aggregate

- Метод Aggregate изпълнява обща агрегация на елементите на колекцията, в зависимост от указания израз. Например:

```
int[] numbers = { 1, 2, 3, 4, 5};
int query = numbers.Aggregate((x,y)=> x - y);
```

- Променлива query ще представлява резултат от агрегацията на масива numbers.
- В качество на условие за агрегация се използва (x,y)=> x - y, тоест в началото от първия елемент се изважда втория, после от получения резултат се изважда третия и така нататък. Тоест, ще бъде еквивалентно на израза:

int query = 1 - 2 - 3 - 4 – 5;

- В резултата се получава -13. Съответно:

int query = numbers.Aggregate((x,y)=> x + y);

//е аналогично на 1 + 2 + 3 + 4 +5;

Получаване на размера на масива. 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 една от версиите си, може да приема ламбда-израз, който задава условие за избор. Ето защо бихме могли да не използваме Where и да напишем

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

Получаване на сума - 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() има редица вариации. В частност, ако имаме набор от сложни обекти, както в примера по-горе, то ние можем да укажем свойство, стойността на което ще се сумира: **users.Sum(n => n.Age)**

Максимална, минимална и средна стойност

- За намиране на минималната стойност се прилага метод Min(), за получаване на максималната - метод Max(), а за намиране на средната - метод Average(). Тяхното действие прилича на методи 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); // минимална възраст

int max1 = numbers.Max();
int max2 = users.Max(n => n.Age); // максимална възраст

double avr1 = numbers.Average();
double avr2 = users.Average(n => n.Age); //средна възраст
```

Методи Skip и Take

- Метод Skip() пропуска определено количество елементи, а метод Take() извлича определено число елементи. Нерядко тези методи се прилагат заедно за създаване на постраничен изход.
- Да извлечем първите три елемента

```
int[] numbers = { -3, -2, -1, 0, 1, 2, 3 };
var result = numbers.Take(3);

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

- Да изберем всички елементи, без първите три:

```
var result = numbers.Skip(3);
```

- Съвместявайки 2-та метода, можем да изберем определено количество елементи, започвайки от определен елемент. Например, да изберем три елемента, започвайки от 5-тия (тоест пропускайки 4 елемента):

```
int[] numbers = { -3, -2, -1, 0, 1, 2, 3 };
var result = numbers.Skip(4).Take(3);
foreach (int i in result)
    Console.WriteLine(i);
```

- По подобен начин работят методи TakeWhile и SkipWhile.
- Метод TakeWhile извлича масив от елементи, започвайки с първия елемент, докато те удовлетворяват определено условие. Например:

```
string[] teams = { "Бавария", "Борусия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона" };
foreach (var t in teams.TakeWhile(x=>x.StartsWith("Б")))
    Console.WriteLine(t);
```

- Съгласно условието избираме тези отбори, които започват с буква Б. В масива има три такива отбора. В цикъла ще бъдат изведени само първите 2:

Бавария

Борусия

- Тъй като 3-тия отбор е "Реал Мадрид" – той не съответства на условието, и след него елементите не се обработват.
- Така, ако първият отбор е "Реал Мадрид", методът ще върне 0 елементи.

- По подобен начин работи и метод SkipWhile. Той пропуска масив елементи, започвайки от първия, докато те удовлетворяват определено условие. Например:

```
string[] teams = { "Бавария", "Борусия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона" };
foreach (var t in teams.SkipWhile(x=>x.StartsWith("Б")))
    Console.WriteLine(t);
```

- Първите два отбора, които започват с буквата Б и съответстват на условието ще бъдат пропуснати. На третия отбор нещата се спират, ето защо последния отбор, започващ с Б, ще бъде изведен:

Реал Мадрид

Манчестер Сити

ПСЖ

Барселона

И ако първия елемен не започва с Б, то метод SkipWhile ще върне всички елементи на масива.

Групиране

- За групиране на данни по определени параметри се прилага оператор **group by** или метод **GroupBy()**. Да допуснем, че имаме масив от обекти от следния тип:

```
class Phone
{  public string Name { get; set; }
   public string Company { get; set; }
}
```

Даденият клас представя модел на телефон, в който са дефинирани определени свойства за названия и компания-производител.

Да групираме телефоните по производител:

```
List<Phone> phones = new List<Phone>
{
new Phone {Name="Lumia 430",
Company="Microsoft" },
new Phone {Name="Mi 5", Company="Xiaomi" },
new Phone {Name="LG G 3", Company="LG" },
new Phone {Name="iPhone 5", Company="Apple" },
new Phone {Name="Lumia 930",
Company="Microsoft" },
new Phone {Name="iPhone 6", Company="Apple" },
new Phone {Name="Lumia 630",
Company="Microsoft" },
 new Phone {Name="LG G 4", Company="LG" }
};
 var phoneGroups = from phone in phones
                   group phone by phone.Company;
 foreach (IGrouping<string, Phone> g in phoneGroups)
{    Console.WriteLine(g.Key);
    foreach (var t in g)
        Console.WriteLine(t.Name);
    Console.WriteLine();
}
```

- Ще получим следния изход:

```
Microsoft
Lumia 430
Lumia 930
Lumia 630
Xiaomi
Mi 5
LG
LG G 3
LG G4
Apple
iPhone 5
iPhone 6
```

- Ако в израз LINQ, като последен оператор, изпълняващ се над набора е group, то операторът select не се прилага.
- Оператор group приема критерий по който се прави групирането: **group phone by phone.Company** - в дадения случай - групиране по свойство Company.
- Резултат от оператора group се явява набор, който се състои от групи. Всяка група представлява обект **IGrouping<string, Phone>:** параметър string указва типа на ключа, а параметър Phone - тип на групираните обекти.
- Всяка група има ключ, който можем да получим чрез свойство Key: g.Key
- Всички елементи на групата може да получим с помощта на допълнителна итерация.
- Елементите на групата имат същия тип, като типа на обектите, които се предават на оператора group, то ест в дадения случай обекти от тип Phone.

- Аналогична заявка може да се построи и с помощта на метода за разширение **GroupBy**:

```
var phoneGroups = phones.GroupBy(p => p.Company);
```

- Сега да изменим заявката и получим команда и създадем от групата нов обект:

```
var phoneGroups2 = from phone in phones
        group phone by phone.Company into g
select new { Name = g.Key, Count = g.Count() };

foreach (var group in phoneGroups2)
    Console.WriteLine("{0} : {1}", group.Name, group.Count);
```

- Изразът

group phone by phone.Company into g

- определя променлива **g**, която ще съхранява групата.
- С помощта на тази променлива ние можем след това да създадем нов обект от анонимен тип:

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

- Сега резултатът от заявката LINQ ще представлява набор от обекти от такива анонимни типове, в които има две свойства Name и Count.
- Резултат от програмата:

Microsoft : 3

Xiaomi : 1

LG : 2

Apple : 2

- Аналогична е операцията с помощта на метода GroupBy():

```
var phoneGroups =
phones.GroupBy(p => p.Company)
.Select(g => new { Name = g.Key, Count = g.Count() });
```

- Сега също можем да направим вложени заявки:

```
var phoneGroups2 = from phone in phones
                   group phone by phone.Company into g
                   select new
                   {
                       Name = g.Key,
                       Count = g.Count(),
                       Phones = from p in g select p
                   };
foreach (var group in phoneGroups2)
{
   Console.WriteLine("{0} : {1}", group.Name, group.Count);
   foreach(Phone phone in group.Phones)
      Console.WriteLine(phone.Name);
   Console.WriteLine();
}
```

- Тук свойство Phones на всяка група се формира с помощта на дополнителна заявка, избираща всички телефони в тази група. Конзолен изход на програмата:

Microsoft : 3
Lumia 430
Lumia 930
Lumia 630

Xiaomi : 1
Mi 5

LG : 2
LG G 3
LG G4

Apple : 2
iPhone 5iPhone 6

- **Аналогичната заявка с помощта на метода GroupBy:**

```
var phoneGroups = phones.GroupBy(p => p.Company)
                .Select(g => new
                {
                   Name = g.Key,
                   Count = g.Count(),
                   Phones = g.Select(p =>p)
                });
```

Съединение на колекции. Метод Join, GroupJoin и Zip

- Съединение в LINQ се използва за обединение на два разнотипни набора в един.
- За съединение се използва оператор join или метод Join().
- Като правило, дадената операция се прилага към 2 набора, които имат **един общ критерий**.
- Например, имаме два класа Player и Team, които имат един общ критерий – **името на отбора**:

```
class Player
{
    public string Name { get; set; }
    public string Team { get; set; }
}
class Team
{
    public string Name { get; set; }
    public string Country { get; set; }
}
```

- Обектите на 2-та класа Team и Player ще имат един общ критерий – **името на отбора**. Да съединим по този критерий двата набора (т.е двете колекции) от обекти на тези класове:

```
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 помощта на израза

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

обект pl от списъка players се съединява с обект t от списъка teams, ако стойността на свойството pl.Team съвпада със стойността на свойството t.Name.

- Резултат от съединението ще бъде обект от анонимен тип, който ще съдържа три свойства. В крайна сметка получаваме следния изход:

Месси - Барселона (Испания)

Неймар - Барселона (Испания)

Роббен - Бавария (Германия)

- Същото действие би могло да се изпълни с помощта метода Join():

```
var result = players.Join(teams,    // втория набор
        p => p.Team,                // свойство-селектор на обекта от първия набор
        t => t.Name,                // свойство-селектор на обекта от втория набор
        (p, t) => new { Name = p.Name, Team = p.Team, Country = t.Country });
                                    // резултат
```

- Метод Join() приема 4 параметъра:

✓ втория списък, който съединяваме с текущия

✓ свойство на обекта от текущия списък, по което става съединението

✓ свойство обекта от втория списък, по което става съединението

✓ новият обект, който се получава в резултат на съединението

GroupJoin

- Освен съединение на последователности, метод GroupJoin изпълнява и групиране. Например, да вземем гореопределените списъци teams и players и да групираме всички играчи по отбора им:

```
var result2 = teams.GroupJoin(
                players, // втория набор
                t => t.Name, // свойство-селектор на обекта от първия набор
                pl => pl.Team, // свойство-селектор на обекта от втория набор
                (team, pls) => new  // резултантния обект
                {
                   Name = team.Name,
                   Country = team.Country,
                   Players = pls.Select(p=>p.Name)
                });

foreach (var team in result2)
{
   Console.WriteLine(team.Name);
   foreach (string player in team.Players)
   {
      Console.WriteLine(player);
   }
   Console.WriteLine();
}
```

//Резултат:

```
Бавария
Роббен

Барселона
Месси
Неймар
```

```
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"}
};
```

- И метод GroupJoin, и метод Join, приемат еднакви параметри.
- Само, че при GroupJoin, на последния параметър - делегат се предават обект от типа на отбора (т.е обект от тип Team) и наборът от играчи на отбора.

Метод Zip

- Метод Zip позволява да се обединят две последователности по такъв начин, че първият елемент от първата последователност се обединява с първия елемент на втората последователност, вторият елемент от първата последователност се съединява с втория елемент на втората последователност и така нататък...

Метод Zip

```
List<Team> teams = new List<Team>()
{
   new Team { Name = "Бавария", Country ="Германия" },
   new Team { Name = "Барселона", Country ="Испания" },
   new Team { Name = "Ювентус", Country ="Италия" }
};
List<Player> players = new List<Player>()
{
   new Player {Name="Роббен", Team="Бавария"},
   new Player {Name="Неймар", Team="Барселона"},
   new Player {Name="Буффон", Team="Ювентус"}
};
var result2 = players.Zip(teams,
                  (player, team) => new
                   {
                       Name = player.Name,
                       Team = team.Name, Country = team.Country
                   });
foreach (var player in result2)
{    Console.WriteLine("{0} - {1} ({2})", player.Name, player.Team,
player.Country);
   Console.WriteLine();
}
```

- В качество на първи параметър метод Zip приема втора последователност, с която трябва да се съедини, а в качество на втори параметър - делегат за създаване на новия обект.
- Конзолен изход от програмата:

Роббен - Бавария (Германия)

Неймар - Барселона (Испания)

Буффон - Ювентус (Италия

Методи All и Any

- Методи All, Any и Contains позволяват да се определи, съответства ли колекцията на определено условие, и в зависимост от резултата връщат true или false.
- Метод All проверява, съответстват ли всички елементи на условието. Например, да узнаем, при всички ли ползватели възрастта превишава 20 и името започва с буква T:

```
List<User> users = new List<User>()
{
    new User { Name = "Tom", Age = 23 },
    new User { Name = "Sam", Age = 43 },
    new User { Name = "Bill", Age = 35 }
};

bool result1 = users.All(u => u.Age > 20);                    // true
if (result1)
Console.WriteLine("При всички ползватели възрастта е по-голяма от 20");
else
Console.WriteLine("Има ползватели с възраст по-малка от 20");

bool result2 = users.All(u => u.Name.StartsWith("T"));        //false
if (result2)
    Console.WriteLine("При всички ползватели името започва с T");
else
    Console.WriteLine("Не при всички ползватели името започва с T");
```

- При всички ползватели възрастта е по-голяма от 20
- Не при всички ползватели името започва с T

- Понеже всички ползватели са с възраст по-голяма от 20, то променлива result1 ще е true. В същото време не всички ползватели са с имена започващи с T, ето защо result2 ще е false.
- Метод Any действа по подобен начин, само позволява да се узнае, съответства ли поне един елемент на колекцията на определено условие:

```
bool result1 = users.Any(u => u.Age < 20); //false
if (result1)
    Console.WriteLine("Има ползватели с възраст по-малка 20");
else
    Console.WriteLine("Всички ползватели са по-големи от 20");

bool result2 = users.Any(u => u.Name.StartsWith("T")); //true
if (result2)
    Console.WriteLine("Има ползватели с име, започващо с T");
else
    Console.WriteLine("Отсъства ползвател с име, започващо с T");
```

- Резултат:

Всички ползватели са по-големи от 20

Има ползватели с име, започващо с T

- Първо изразът ще върне false, доколкото всички потребители са по-големи от 20. Второ изразът ще върне true, тъй като в колекцията има ползвател с име Tom.

Отложено и незабавно изпълнение на LINQ

- Има два начина за изпълнение на заявка LINQ: **отложено** и незабавно изпълнение.
- При отложеното изпълнение LINQ-изразът не се изпълнява, докато не бъде произведена итерация или преглед на извадката. Да разгледаме отложеното изпълнение:

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

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

// изпълнение LINQ-заявка
foreach (string s in selectedTeams)
    Console.WriteLine(s);
```

- Тоест, фактическото изпълнение на заявката ще стане не в реда:
- var selectedTeams = from t..., а при изброяването в цикъла foreach. За да се види по-нагледно това, ние можем да изменим някой елемент до обхождането на извадката.

```
var selectedTeams = from t in teams where
t.ToUpper().StartsWith("Б") orderby t select t;
// изменение на масива след дефиниране на LINQ-заявка
teams[1] = "Ювентус";
// изпълнение на LINQ-заявка
foreach (string s in selectedTeams)
    Console.WriteLine(s);
```

- Сега наборът ще съдържа два елемента, а не три, тъй като вторият елемент след изменението няма да съответства на условието.
- Ако в LINQ-заявка се използват методи за разширение, които връщат резултат, отличаващ се от типа на последователността, например, методи ToArray<T>(), ToList<T>(), Count() и т.д.. Например:

```
string[] teams = {"Бавария", "Борусия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона"};
// изпълнение на LINQ-заявка
var selectedTeams = (from t in teams
                     where t.ToUpper().StartsWith("Б")
                     orderby t select t).ToList<string>();
/* изменението на масива никак не отразява на списъка selectedTeams */
teams[1] = "Ювентус";

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

- С помощта на метода ToList() се изпълнява преобразование на последователността към тип List<string>, ето защо ще се приложи незабавно изпълнение. Ето защо, вън от зависимостта на това, ще се изменя ли масив teams или не, заявката списък selectedTeams ще съдържа три елемента.
- Да разгледаме още и пример с метода Count(), който връща броя на елементите в последователността:

```
string[] teams = {"Бавария", "Борусия", "Реал Мадрид",
"Манчестер Сити", "ПСЖ", "Барселона"};
// изпълнение на LINQ-заявка
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
```

- Резултат от метода Count ще бъде обект int, който така също се отличава от типа на последователността, ето защо сработва незабавно изпълнение.
- Но може да се измени кода така, че метод Count() да отчете измененията:

```
string[] teams = {"Бавария", "Борусия", "Реал Мадрид",
"Манчестер Сити", "ПСЖ", "Барселона"};
// изпълнение LINQ-заявка
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
```

Делегати и анонимни методи в заявки LINQ

- Като правило, в качество на параметри в методите за разширение на LINQ е удобно да се използват ламбда-изрази.
- Но ламбда-изразите се явяват съкращения на нотациите на анонимните методи. И ако се обърнем към дефинициите на тези методи, то ще видим, че в качество на параметър много от тях приемат делегати от типа Func<TSource, bool>, например, дефиницията на метода Where:

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

- Да зададем параметри чрез делегати:

```
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);
```

- Тък като наборът от елементи, към които се прилага метод Where, съдържа обекти int, то в делегата в качеството на параметър се предава обект на този тип. Връщаният тип трябва да е тип bool: ако е true, то обект int соответства на условието и попада в извадката.

- Алтернативният подход представлява поместване на цялата логика в отделен 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;
}
```

- Да разгледаме друг пример. Нека метод Select() добавя в извадката не текущия елемент-число, а неговия факториел.

```
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();
 }
```

Метод Select в качество на параметър приема тип Func<TSource, TResult> selector. Тъй като при нас наборът от обекти е int, то входните параметри на делегата също ще бъдат обекти от типа int. В качеството на тип на изходния параметър избираме int, тъй като факториелът на едно число е с целочислена стойност.

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

Използвана литература

- http://metanit.com/sharp/tutorial/15.1.php