

HELP LINQ команди

* За примерите ще използваме два списъка
– един с прост тип данни (цели числа) и един от сложен тип данни Phone

```
List<int> nums = new List<int> {10, 15, 7, -5, 33, 50, -27};
```

```
List<Phone> phones = new List<Phone> {...};
```

Company	Model	Year	OS	Amount	BuyPrice	SellPrice
Apple	iPhone 7	2016	iOS	42	245	350
Apple	iPhone 8	2017	iOS	36	280	400
Apple	iPhone XR	2018	iOS	37	350	500
Nokia	6.2	2018	Android	59	154	220
Apple	iPhone 11	2019	iOS	24	462	660
Apple	iPhone 11 Pro	2019	iOS	58	805	1150
Nokia	7.2	2019	Android	65	175	250
Nokia	2.3	2019	Android	15	84	120
Nokia	9 PureView	2019	Android	14	455	650
Sony	Xperia 10	2019	Android	15	280	400
Apple	iPhone 12	2020	iOS	45	490	700
Apple	iPhone SE	2020	iOS	51	280	400
Nokia	3.4	2020	Android	53	112	160
Sony	Xperia 5	2021	Android	25	560	800

Select(e => преобразуване)

Идеята е да обходи всички елементи от колекцията, която се образува преди него, като ги преобразува до нов вид елементи. Т.е. срещу всеки елемент от първоначалната колекция образува нов, като не е задължително новия да бъде от същия тип!

Примери:

```
List<int> newNums = nums.Select(e => (e + 5)).ToList(); //ще върне списък с числа по-големи с 5
```

```
var newNums = nums.Select(p => 2*p).ToArray(); //ще върне масив с числата умножени по 2
```

*** VAR сам определя типът на данните, като го приравнява на типът след всички преобразования в дясно!!!

Затова и е най-удачно да се ползва при преобразования с LINQ

```
var newNums = nums.Select((p,i) => i).ToArray(); //тук интересното е, че ако използвате два параметъра от колекция, то първия приема елемента, а втория индекса от колекцията. Съответно това би върнало масив от индексите на колекцията, т.е. числата от 0 до дължината на минус 1 колекцията
```

```
var newNums = nums.Select(p => $"{p:N2}лв." ).ToList(); //ще преобразува числата до съответния текст
```

```
10.00лв.  
15.00лв.  
7.00лв.  
-5.00лв.  
33.00лв.  
50.00лв.  
-27.00лв.
```

Where(e => **Predicat(e)**)

- Обхожда всички елементи от някаква колекция и допуска напред само елементи по някакво конкретно условие (функция която връща истина или лъжа – наричаме ги Предикати). Ако условието е вярно елемента се „пропуска/участва” в новата колекция, ако не е то елемента не се допуска в нея.

Примери:

```
var newNums = nums.Where(e => true).ToList(); //би пропуснало всички, т.е. ще имате копие на първоначалния списък
```

```
var newNums = nums.Where(e => e%2==0).ToList(); //ще пропусне само четните числа
```

```
var newNums = nums.Where((e,i) => i%2==0).ToList(); //би пропуснало само четните индекси от първоначалния списък
```

```
var newNums = nums.Where((e,i) => i<5).ToList(); //би пропуснало само първите 5 елемента
```

```
/* равносилно на var newNums = nums.Take(5).ToList();
```

```
var newNums = nums.Where(e => e>10).ToList(); //би пропуснало числата по-големи от 10
```

```
var newPhones = phones.Where(p => p.Company.Substring(0,1)=="N").ToList(); //ще пропусне само телефоните чийто Бранд започва с първа Буква 'N'
```

в тялото на Ламбда функциите можем да описваме повече от една операция. Тогава се нуждаем от фигурни скоби и RETURN

```
var newPhones = phones.Where(p => {  
    if(p.Company.Substring(0,1)=="N" && p.SellPrice>500) return true;  
    return false;  
}).ToList(); //ще пропусне само Nokia 9 PureView понеже е с цена 650
```

```
//От всеки един Бранд искаме модела с най-много продажби  
items.GroupBy(p => p.Company)  
    .ToList()  
    .ForEach(gr => {  
        Console.WriteLine($"Бранд:{gr.Key} Най-продаван модел:");  
        gr.Where(p => p.Amount == gr.Max(ph => ph.Amount))  
            .ToList()  
            .ForEach(p =>  
                Console.WriteLine(p.Model + " " + p.Amount + "броя"));  
    });
```

```
Бранд:Samsung Най-продаван модел:Galaxy S10 57броя  
Бранд:Apple Най-продаван модел:iPhone 11 Pro 58броя  
Бранд:Huawei Най-продаван модел:P40 32броя  
Бранд:Xiaomi Най-продаван модел:Redmi Note 9 67броя  
Бранд:Sony Най-продаван модел:Xperia Z5 64броя  
Бранд:Nokia Най-продаван модел:7.2 65броя  
Бранд:LG Най-продаван модел:G7 ThinQ 61броя
```

* тук обърнете внимание, че **gr** е обект от тип **IGrouping<string,GSM>** т.е. съдържа в себе си колекция, а **Where** пропуска елементи от колекцията към която го използваме

** в едни и същи скоби не позволява дублиране на имена, затова използваме както **p**, така и **ph** за инстанциите на телефоните

ForEach(e => Действие/я(e))

Внимание! Това е екстеншън метод само за класа `List<T>` т.е. не е част от LINQ, но се комбинира добре с LINQ методите

Обхожда всички елементи от Списъка и за всеки един изпълнява някаква конкретна задача. Най-често ще го използваме за печатане на данни в конзолата. Имайте предвид, че този метод е void! Т.е. извършват се някакви действия, но не трябва да връща резултат. Което автоматично означава, че след него няма как да добавяме други действия.

```
void List<string>.ForEach(Action<string> action)  
Performs the specified action on each element of the List<T>.
```

ToList()

✚ Преобразува обходима колекция до списък.

Ако искаме да използваме екстеншън метода ForEach на класа List<T>, а към момента имаме някакъв изброим тип данни, то това е решението

OrderBy(e => критерий)

✚ сортира IEnumerable елементи във възходящ ред или азбучен ред

OrderByDescending (e => критерий)

✚ сортира IEnumerable елементи в нисходящ или обратно на азбучен ред

ThenBy(e => критерий) и ThenByDescending(e => критерий)

✚ аналогични на по-горните, но се използват са втори, трети и т.н. критерий за сортирането

GroupBy(e => e.field)

✚ групира сложни обекти (т.е. имащи полета) по дадено поле

Примерно ако в ППМГ випуск 2025 има 100 човека, а всеки клас е от 25 души. И всеки ученик има полета `ClassLetter`, `ClassVipusk`, `NumInClass`, `FirstName`, `LastName` и т.н. то ако групираме `GroupBy(student => student.ClassLetter)` ще се получат 4 нови обекта

```
var classes = students.GroupBy(st => st.ClassLetter);
```

```
(local variable) IEnumerable<IGrouping<string, Student>> classes
```

всеки от които ще има за ключ – буквата на класа, а в него ще има изброима колекция от Student. Самата колекция от 4 елемента е тип IEnumerable – т.е. изброим тип, но много лесно може да си я превърнете в List, чрез `.ToList()` в края на кода.

Аналогично е на Dictionary<key, TValue> с единствената разлика, че тук вместо TValue имаме IEnumerable колекция

За значително повече информация при работа с GroupBy() отворете файла с решените 12 задачи.

Min(), Max(), Sum(), Average(), Count()

✚ връщат агрегиран резултат от цяла колекция като може да е от прост тип данни, а може и да е от сложен тип данни

Примерно:

```
List<int> nums = new List<int> {10, 15, 7, -5, 33, 50, -27};
```

- `nums.Min()` би върнало резултат -27
- `nums.Count()` ще върне резултат 7

А когато искаме да приложим върху поле от клас се указва чрез полето на класа

- `phones.Max(ph => ph.SellPrice)` ще върне резултат 1150

Ако искаме да извлечем най-скъпия модел, освен чрез сортиране и вземане на най-горния елемент `Take(1)` то можем да направим и ето така:

```
var maxS = phones.Max(ph => ph.SellPrice);

phones.Where(ph => ph.SellPrice == maxS)
    .Select(ph => $"Бранд:{ph.Company} Модел:{ph.Model} Пр.цена:
    {ph.SellPrice}") .ToList().ForEach(s => Console.WriteLine(s));
```

Още веднъж по същина:

Select() – извършва преобразуване на входните данни! Очаква действия връщащи резултат (`Func<T..., TResult>`)

Следователно в него може да вмъкваме директно преобразувания, ламбди, `Func<>` или невоид методи!

Where() – извършва филтриране по критерий. Очаква действия връщащи `bool` (`Func<T..., bool>`)

Следователно в него може да извършваме преобразувания връщащи `bool`, ламбда връщаща `bool`, `Func<T..., bool>`, `Predicate(T)` или метод връщащ `bool`.

ForEach() – екстенд метод на `List<>`. Очаква само действия, които не връщат резултат. Т.е. това би било последното действие при обработка на дадена колекция.

Относно `Func<>`, `Action<>`, `Predicate<>` вижте подробностите на следващите 2 страници.

Относно Func<T, TResult>, Action<T> и Predicate<T>

И трите почти по никакъв начин не се различава от обикновенните методи. Но имат някои особености:

Върнатият резултат

- ✚ Func<> винаги връща резултат (и това е последният тип в диамантените скоби)
- ✚ Action<> никога не връща резултат (т.е. той е void)
- ✚ Predicate<> винаги връща bool (булев тип истина/лъжа)

И трите действия създават делегати (ако обърнете внимание синтаксиса е светлосин)

Входни данни

- ✚ Типът на входните данни се задава в диамантените скоби, като единствено при Func последният тип данни е на върнатият резултат, а не на входните данни (параметри)

И четирите кода от дясно извършват едно и също действие.

- 1) Използвано стандартно описание на метод + return за върнатия тип данни,
- 2) Използван е съкратен запис чрез ламбда функция за върнатия тип данни
- 3) Използван е Func<> което създава делегат, който връща булев тип данни
- 4) Използван е Predicate<> което е частен случай на делегат, който връща булев тип данни

```
static bool GetUnder5(string txt)
{
    return txt.Length < 5;
}
1
2 static bool GetUnder5(string txt) => txt.Length < 5;
3 Func<string, bool> getUnder5 = txt => txt.Length < 5;
4 Predicate<string> getUnder5 = txt => txt.Length < 5;
```

Пример преобразуващ подадените числа към стринг:

```
Func<int, string> converterIntToString = x => x.ToString();
string five = converterIntToString(5);
```

Пример за използване на делегата като действие в LINQ команда **изисквам да се знае в раздела за 6-ца**

```
Func<int, int> getSquare = n => n * n; //ще отпечата квадратите
nums.Select(getSquare).ToList().ForEach(n => { Console.WriteLine(n); });
```

Едно малко уточнение: И Select и Where очакват делегат от тип Func<>, като в частност Where очаква Func<T,bool> Затова и при вмъкване в скобите, ако използваме функция-променлива създадена чрез Func<> не е необходимо да подаваме изрично в скоби параметъра. Докато ако използваме Predicate<> то за да сработи синтаксисът трябва изрично да подадем в скоби и самия параметър

```
Func<GSM, bool> brandFilter1 = br => br.Company.Length <= 5;
items.Where(brandFilter1).ToList();

Predicate<GSM> brandFilter2 = br => br.Company.Length <= 5;
items.Where(ph => brandFilter2(ph)).ToList();
```

Action<> е аналогично на Func<>, но просто не връща резултат. Затова и екстенд метода на Лист - ForEach() реално изисква Action<>

```
Action<string> write = txt => Console.WriteLine(txt);
Action<int> upAndPrint = num => { num++; Console.WriteLine(num); };
List<int> nums = new List<int>() { 10, 15, 7, -5, 33, 50, -27 };
nums.ForEach(upAndPrint);
```

```
void List<int>.ForEach(Action<int> action)
Performs the specified action on each element of the List<T>.
```

Разбира се може да се използва и външен метод вместо Екшън функция-променлива. Ето аналогичното решение:

```
static void UpAndPrint(int num)
{
    num++;
    Console.WriteLine(num);
}

а това бихме го използваме ето така:
nums.ForEach(n => UpAndPrint(n));
```