

Zawartość:

- First/FirstOrDefault/Last/LastOrDefault
- Where
- ForEach
- Select
- Contains
- Remove
- Max/Min
- ToList()/ ToArray()
- mała wzmianka o bazach danych i ORM

LINQ – Language Integrated Query

Najprościej rzecz ujmując jest to biblioteka pod język C# służąca do pisania zapytań do obiektów oraz kolekcji.

Przykładowe zastosowania:

- chcemy wybrać z listy/słownika/tablicy pierwszy element lub wszystkie elementy pasujące do kryterium
- chcemy z listy/słownika/tablicy uzyskać podzbiór elementów pasujących do kryterium
- chcemy posortować dane według kryterium
- chcemy wysłać polecenie do bazy danych z wykorzystaniem ORMa takiego jak Entity Framework
- i wiele więcej

Zapytania LINQ można w C# pisać na dwa sposoby.

Pierwszym z nich jest zwykły zapis metoda po metodzie „chaining”:

```
List<int> myList = new List<int>() { 1,6,5,4,3,2,1,2,3,5};  
List<int> myList2 = myList.Where(number => number > 3).ToList();
```

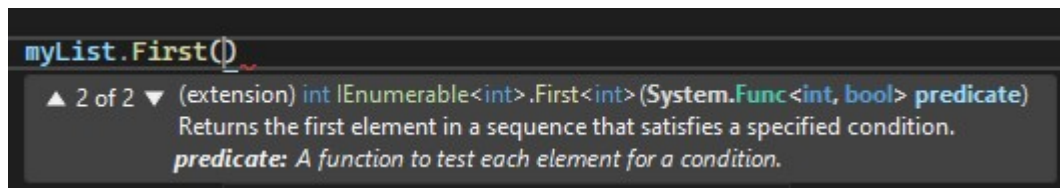
Drugi z nich przypomina bardziej zapytania SQLowe:

```
List<int> myList = new List<int>() { 1,6,5,4,3,2,1,2,3,5};  
var myList3 = from number in myList where number > 3 select number;
```

Rozwiązanie numer jeden jest znacznie popularniejsze.

First(), FirstOrDefault(), Last(), LastOrDefault()

Metody te zwracają pierwszy lub ostatni element pasujący do podanego warunku.



Jak widać metody te występują w dwóch wersjach:

- Bez parametru – zwracają one wtedy pierwszy lub ostatni element
- Z parametrem w formie funkcji przyjmującej argument typu T oraz zwracającej boola (myList przechowuje inty i dlatego pierwszy parametr generyczny to int (Func<int, bool>))

Czym się różni First() od FirstOrDefault()?

Jeżeli nie będzie możliwe zwrócenie elementu ponieważ kolekcja jest pusta lub nie znaleziono dopasowania to funkcja First() rzuci wyjątek (będziemy mieli błąd) a funkcja FirstOrDefault() zwróci domyślną wartość danego typu. Jeżeli mamy listę intów to domyślną wartością jest 0, jeżeli mamy listę obiektów jakiejś klasy/struktury to domyślną wartością jest null.

Last() oraz LastOrDefault() działają identycznie.

```
List<int> myList = new List<int>() { 1,6,5,4,3,2,1,2,3,5};  
  
var a = myList.First(); //zwróci 1  
var b = myList.FirstOrDefault(); //zwróci 1  
  
var c = myList.First(n => n>1); //zwróci 6  
var d = myList.FirstOrDefault(n => n>1); //zwróci 6  
  
//var e = myList.First(n => n>10); //error  
var f = myList.FirstOrDefault(n => n>10); //0
```

Jeżeli funkcja podana do parametru zwróci prawdę element jest brany pod uwagę. Funkcję podawane jako parametry metod LINQ to zazwyczaj lambdy, ale nikt nie zabrania podania tam czegoś bardziej skompikowanego. Ważne by metoda podana w parametrze pasowała do delegata Func<T, bool> gdzie T to typ elementów listy/słownika/tablicy.

Inny przykład dla tablicy obiektów klasy Person:

```
5 references
class Person
{
    3 references
    public string Name { get; set; }
    3 references
    public string Surname { get; set; }
    3 references
    public int Age { get; set; }
}
```

```
List<Person> myList = new List<Person>() {
    new Person() {Name="Jan", Surname="Kowalski", Age=20},
    new Person() {Name="Adam", Surname="Nowak", Age=17}
};
myList.Add(new Person() { Name = "Justyna", Surname = "Adamowicz", Age = 18 });

var a = myList.First(); //zwróci Jana Kowalskiego
var b = myList.FirstOrDefault(); //zwróci Jana Kowalskiego

var c = myList.First(p => p.Age < 18); //zwróci Adama
var d = myList.FirstOrDefault(p=>p.Age < 18); //zwróci Adama

//var e = myList.First(p=>p.Age>30); //error
var f = myList.FirstOrDefault(p=>p.Age>30); //zwróci null
```

Wyniki:

myList	Count = 3
a	{ConsoleApp1.Program.Person}
Age	20
Name	"Jan"
Surname	"Kowalski"
b	{ConsoleApp1.Program.Person}
Age	20
Name	"Jan"
Surname	"Kowalski"
c	{ConsoleApp1.Program.Person}
Age	17
Name	"Adam"
Surname	"Nowak"
d	{ConsoleApp1.Program.Person}
Age	17
Name	"Adam"
Surname	"Nowak"
f	null

Co warto zauważyć to to, że podpowiedzi dopasowały się do typu elementów w liście:

```
myList.FirstOrDefault()
```

▲ 2 of 2 ▼ (extension) **Person** IEnumerable<**Person**>.FirstOrDefault<**Person**>(**System.Func**<**Person**, **bool**> predicate)
Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
predicate: A function to test each element for a condition.

Jak widać parametr lambdy lub funkcji to Person (zmienna p w powyższym przykładzie). Typ zwracany to dalej bool.

Where()

Funkcja ta służy do filtrowania kolekcji podobnie jak te w poprzednim punkcie lecz ta nie zatrzymuje się po znalezieniu jednego elementu. Zwraca ona wszystkie elementy spełniające warunek.

```
myList.Where(x => Compare(x));  
//zwróci 1 (extension) IEnumerable<int> IEnumerable<int>.Where<int>(System.Func<int, bool> predicate) (
```

Trzeba tutaj zwrócić uwagę co zwraca funkcja Where(). Jak widać zwraca ona IEnumerable<T> czyli typ bazowy z którego dziedziczy lista. Aby można było przypisać wynik działania Where() do listy należy posłużyć się metodą ToList().

```
List<int> myList = new List<int>() { 1, 6, 5, 4, 3, 2, 1, 2, 3, 5 };  
  
IEnumerable<int> result1 = myList.Where(x => Compare(x));  
List<int> result2 = myList.Where(x => Compare(x)).ToList();
```

IEnumerable to typ bazowy dla Listy (jest to interfejs). Lista rozbudowuje go o parę dodatkowych możliwości.

Jak widać w powyższym przykładzie posłużyliśmy się dodatkową funkcją zamiast pisać wszystko w wyrażeniu lambda. Ma to sens jeżeli chcemy tam upchnąć trochę więcej logiki (czytelność). Tutaj wykonaliśmy prosty test czy element równy jest 1.

```
2 references  
private static bool Compare(int x)  
{  
    return x == 1;  
}
```

To samo możemy wykonać dla listy z obiektami klasy Person.

```
List<Person> myList = new List<Person>() {  
    new Person() {Name="Jan", Surname="Kowalski", Age=20},  
    new Person() {Name="Adam", Surname="Nowak", Age=17}  
};  
myList.Add(new Person() { Name = "Justyna", Surname = "Adamowicz", Age = 18 });  
  
IEnumerable<Person> result1 = myList.Where(x => x.Surname.StartsWith('N'));  
List<Person> result2 = myList.Where(x => x.Surname.StartsWith('N')).ToList();
```

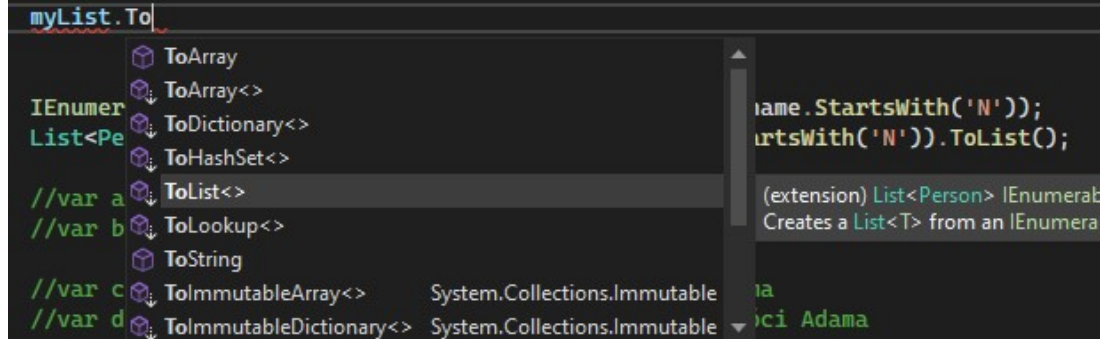
Zastosowano tutaj trochę inny rodzaj filtra.

Metody z serii ToX()

LINQ pozwala konwertować kolekcje z typu na typ. Np. z tablicy do listy. Z listy do tablicy itd:

```
List<Person> myList = new List<Person>() {
    new Person() {Name="Jan", Surname="Kowalski", Age=20},
    new Person() {Name="Adam", Surname="Nowak", Age=17}
};
myList.Add(new Person() { Name = "Justyna", Surname = "Adamowicz", Age = 18 });

myList.To
```



Dostępne opcje widać w podpowiedziach. Oczywiście to kosztuje. Czasami w kodzie C# widać jak ktoś co linie wykonuje konwersje bo akurat tym razem była mu potrzebna metoda z tego typu kolekcji a raz z innego. Takie coś zabija wydajność. Starajmy się trzymać za wszelką cenę jedną wersję kolekcji.

Sortowanie za pomocą OrderBy(), OrderByDescending()

```
List<Person> myList = new List<Person>() {
    new Person() {Name="Jan", Surname="Kowalski", Age=20},
    new Person() {Name="Adam", Surname="Nowak", Age=17}
};
myList.Add(new Person() { Name = "Justyna", Surname = "Adamowicz", Age = 18 });

List<Person> result3 = myList.OrderBy(p=>p.Surname).ToList();
List<Person> result4 = myList.OrderByDescending(p=>p.Surname).ToList();
```

Wyrażenie podane w parametrze zwraca na bazie jakiej wartości obiektu ma zostać wykonane sortowanie. W przypadku intów nie ma tego dylematu:

```
List<int> myList = new List<int>() { 1, 6, 5, 4, 3, 2, 1, 2, 3, 5 };

List<int> result3 = myList.OrderBy(n=>n).ToList();
List<int> result4 = myList.OrderByDescending(n=>n).ToList();
```

ForEach

Znany patent także z JavaScriptu czyli skrótowa funkcja iterująca po kolekcji:

```
myList.ForEach(myList.Add);  
  
void List<int>.ForEach(System.Action<int> action)  
Performs the specified action on each element of the List<T>.  
action: The System.Action<in T> delegate to perform on each element of the List<T>.
```

Jak widać dla listy intów oczekuje ona Action<int> czyli lambdy/metody z jednym parametrem typu int, która zwraca void.

```
List<int> myList = new List<int>() { 1, 6, 5, 4, 3, 2, 1, 2, 3, 5 };  
myList.ForEach(i => System.Console.WriteLine(i));
```

Takie coś wypisze po prostu wszystkie elementy. Pytanie co stanie się w przypadku kodu poniżej:

```
myList.ForEach(i => i+=1);  
myList.ForEach(i => System.Console.WriteLine(i));
```

Zostaną wypisane elementy oryginalne (nie zwiększone o 1). Dlaczego? Typ int jest typem wartościowym. Przekazując go do lambdy (która jest funkcją) w parametrze ta wykonuje kopię elementu i zwiększa tę kopie. Funkcja się kończy, kopia zostaje zgubiona bo nic z nią nie zrobiliśmy a lista zostaje bez zmian. Inaczej sprawa ma się tutaj:

```
List<Person> myList = new List<Person>() {  
    new Person() {Name="Jan", Surname="Kowalski", Age=20},  
    new Person() {Name="Adam", Surname="Nowak", Age=17}  
};  
myList.Add(new Person() { Name = "Justyna", Surname = "Adamowicz", Age = 18 });  
  
myList.ForEach(i => i.Age += 1);  
myList.ForEach(i => System.Console.WriteLine($"{i.Name} {i.Surname} {i.Age}"));
```

Tutaj zobaczymy wiek zwiększony o 1. Dlaczego? Klasa to definicja typu referencyjnego. Typ referencyjny jest przekazywany do funkcji przez referencję (nie jest tworzona kopia tego obiektu lecz funkcja operuje na oryginale).

Select()

Select to metoda, która pozwala wykonać swego rodzaju remapowanie wartości. Mówiąc prostszym językiem pozwala wybrać wszystko lub tylko część wartości z obiektów i zapisać pod nowym obiektem:

```
List<int> myList = new List<int>() { 1, 6, 5, 4, 3, 2, 1, 2, 3, 5 };

//lista obiektów anonimowych
var result5 = myList.Select(x => new {Desc="Opisik", Value=x}).ToList();
//lista obiektów jakiejś klasy
var result6 = myList.Select(x => new Item{ Desc = "Opisik", Value = x }).ToList();
```

Dla każdego inta w oryginalnej liście stworzyliśmy dwie listy. Druga z nich to zwykłe mapowanie na obiekt klasy Item (utworzona wyżej). Prosta sprawa. Opis dodany dla urozmaicenia. W pierwszym przypadku tworzymy tak zwany typ anonimowy. Odwoływać się możemy do niego w następujący sposób:

```
//tu zwracamy liste obiektów anonimowych
var result5 = myList.Select(x => new {Desc="Opisik", Value=x}).ToList();

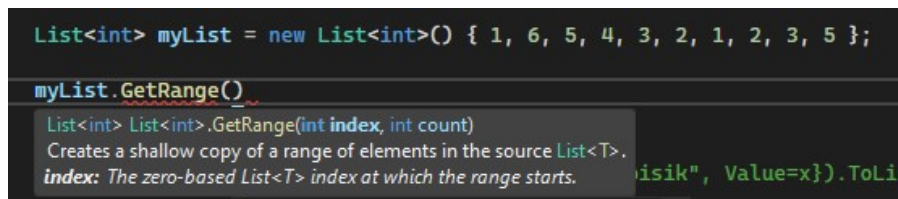
//tu zwracamy pierwszy z wygenerowanych obiektów
var result6 = myList.Select(x => new {Desc="Opisik", Value=x}).First();

System.Console.WriteLine(result5[0].Value);
System.Console.WriteLine(result6.Value);
```

Jak widać VS pamięta z jakich elementów składa się nasz typ anonimowy mimo, że nie tworzyliśmy klasy czy struktury (tej foremki na te dane). Po co to? Żeby nie tworzyć na siłę klas (foremek na dane), które są potrzebne raz na parę chwil.

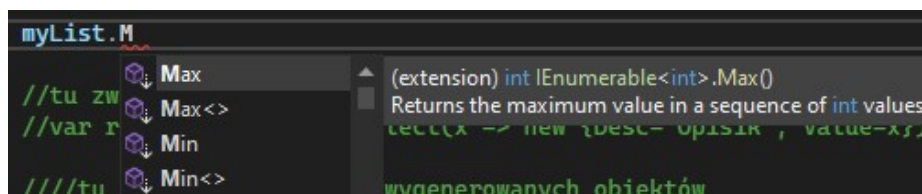
Inne przydatne metody:

GetRange() – zwraca fragment kolekcji



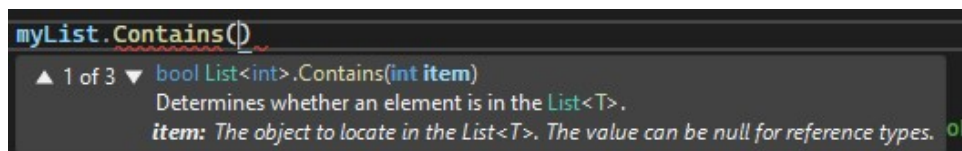
Jak widać podajemy tej funkcji indeks oraz ilość elementów do pobrania. Zwraca ona `List<T>`.

Min(), Max() – tutaj nie trzeba nic tłumaczyć

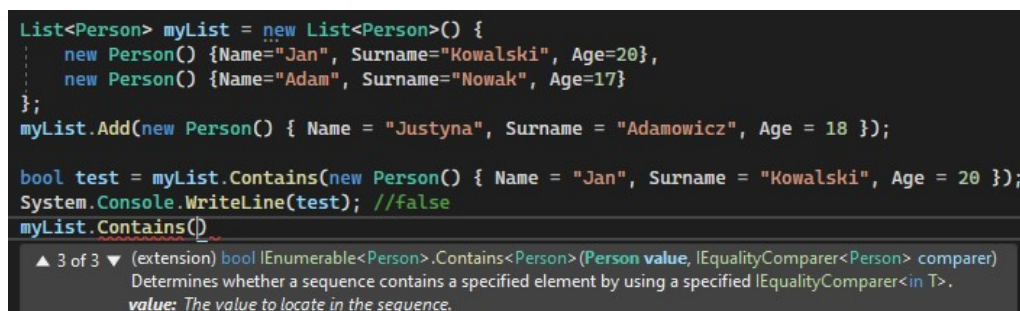


Brak parametrów. Zwracana jest jedna sztuka obiektu typu T.

Contains() – zwraca true/false zależnie od tego czy kolekcja zawiera dany element



Niby prosta sprawa. Ale w przypadku typów referencyjnych nie jest tak kolorowo:



Referencja to w teorii wskaźnik. Contains() zwróciło false bo wskaźniki są różne. W powyższym fragmencie kodu zrobiliśmy dwa obiekty Jana Kowalskiego o wieku 20. Dokładnie dwa. Oba są na innych adresach w pamięci. Contains porównał adres i wyszło mu, że są różne.

Aby to zadziałało trzeba funkcji podać specjalny obiekt klasy porównującej. A taką klasę trzeba sobie stworzyć samemu na bazie interfejsu `IEqualityComparer<T>`.

Ale ale uwaga:

```
List<Person> myList = new List<Person>() {
    new Person() {Name="Jan", Surname="Kowalski", Age=20},
    new Person() {Name="Adam", Surname="Nowak", Age=17}
};
Person p = new Person() { Name = "Justyna", Surname = "Adamowicz", Age = 18 };
myList.Add(p);

bool test = myList.Contains(p);
System.Console.WriteLine(test); //true
```

Tutaj wychodzi prawda, bo do `Contains()` podrzucamy tę samą referencję (ten sam adres), który dodaliśmy do listy. Tu i tu odnosimy się do tego samego obiektu – tego samego miejsca w pamięci.

`Remove()` – funkcja usuwająca element z kolekcji. Dla typów wartościowych sprawa prosta. ALE dla typów referencyjnych zachodzi taka sama sytuacja jak w przypadku `Contains()`. `Remove` musi najpierw za pomocą prawdopodobnie `Contains()` stwierdzić czy kolekcja faktycznie zawiera element, który chcemy usunąć więc znowu musi wykonać porównania.

```
List<Person> myList = new List<Person>() {
    new Person() {Name="Jan", Surname="Kowalski", Age=20},
    new Person() {Name="Adam", Surname="Nowak", Age=17}
};
Person p = new Person() { Name = "Justyna", Surname = "Adamowicz", Age = 18 };
myList.Add(p);

myList.Remove(p);
```

<code>Remove</code>	<code>bool List<Person>.Remove(Person item)</code>
<code>RemoveAll</code>	<code>Removes the first occurrence of a specific object from the List<T>.</code>

Jak widać zwracany jest `bool` z informacją czy udało się usunąć element. W powyższym przypadku będzie to prawda.

`Insert()` – wstawia element na podany indeks – pozostałe elementy są przesuwane dalej

```
List<Person> myList = new List<Person>() {
    new Person() {Name="Jan", Surname="Kowalski", Age=20},
    new Person() {Name="Adam", Surname="Nowak", Age=17}
};
Person p = new Person() { Name = "Justyna", Surname = "Adamowicz", Age = 18 };

myList.Insert(0, p); //wstaw p na index 0
```

Mała wzmianka o ORMach

ORM to biblioteka, która wykonuje za nas zapytania do bazy danych. Mapuje ona wyniki zapytań na obiekty i odwrotnie. Przykład:

```
namespace Astro.DAL.Models
{
    public class Topic
    {
        public int Id { get; set; }

        public int Rate { get; set; }

        public string Title { get; set; }

        public string Date { get; set; }

        public User User { get; set; }

        public List<Comment> Comments { get; set; }
    }
}
```

Tutaj widzimy przykładową klasę Topic. Na nią będą mapowane zapytania do tabeli o tej samej nazwie. Klasa ta ma pole User, które jest mapowane na powiązanego z tym tematem użytkownika w tabeli User (relacja jeden do wielu). Mamy też listę komentarzy (relacja wiele do jednego).

Tak wygląda konfiguracja Entity Framework (jedyne sensowny ORM dla C#).

```
public class AstroDbContext : IdentityDbContext<User>
{
    public AstroDbContext(DbContextOptions options) : base(options)
    {
    }

    public DbSet<Comment> Comments { get; set; }
    public DbSet<Topic> Topics { get; set; }
    public DbSet<User> User { get; set; }
    public DbSet<APOD> APOD { get; set; }
    public DbSet<EPIC> EPIC { get; set; }
    public DbSet<AsteroidsNeOWs> AsteroidsNeOWs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<Topic>()
            .HasOne(p => p.User)
            .WithMany(b => b.Topics);

        modelBuilder.Entity<Comment>()
            .HasOne(p => p.Topic)
            .WithMany(b => b.Comments);

        modelBuilder.Entity<Comment>()
            .HasOne(p => p.User)
            .WithMany(p => p.Comments);
    }
}
```

Znajduje się tu lista tabel w formie właściwości. Oraz jedna metoda konfiguracyjna mówiąca jak połączyć klucze obce itp.

Wspominam o tym, ponieważ zapytania pisze się tak:

```
public async Task<IActionResult> MainPage()
{
    List<Topic> topics = await _context.Topics.AsNoTracking().Include(t => t.User)
        .Include(t=>t.Comments).OrderByDescending(t => t.Id)
        .ToListAsync();

    return View(topics);
}
```

Jak widać mamy tu do czynienia z LINQ. Obiekt `_context` (1) to instancja tej klasy konfiguracyjnej ze screena wyżej. `Topics` to jej właściwość (2). (3) służy to tego by ORM przestał śledzić to co robimy z tymi pobranymi obiektami. Inaczej monitoruje on każdą zmianę w obiektach i zapisuje do bazy (w tym przypadku robimy read-only). (4) mówi ORMowi by wczytał dane będące w relacji z tą tabelą (nie zawsze musimy tego chcieć bo po co ciągnąć dane z bazy bez potrzeby jak ich nie potrzebujemy).

```
Topic topic = await _context.Topics.FirstOrDefaultAsync(t => t.Id == id);

if (up)
    topic.Rate++;
else
    topic.Rate--;

await _context.SaveChangesAsync();
```

Tutaj pobieramy jeden temat o konkretnym ID. Aktualizujemy mu pole i zapisujemy w bazie zmiany. Używamy LINQ zamiast SQL. Tak czy siak na potrzeby aplikacji musielibyśmy tworzyć obiekty z wyników zwróconych przez zapytanie. ORM ma tu przewagę, ponieważ robi za nas bardzo dużo.