

kursdotnet.pl

LINQ

Podstawy skutecznej
pracy z danymi
w języku C#



Dariusz Kacban

LINQ

Podstawy skutecznej pracy z danymi w języku C#

Dariusz Kacban

Wstęp

Powszechnie uważa się, że programistą C# można zostać bardzo łatwo - wystarczy opanować zmienne, instrukcje warunkowe, pętle i już można zacząć pracę. Brzmi świetnie, ale w praktyce to nie jest jednak takie proste.

Pamiętam kiedy zaczynałem swoją pierwszą pracę jako programista .NET. Język C# znałem jedynie z książek takich jak *Microsoft Visual C# krok po kroku*. Na studiach poznałem składnię języka C#. Ale to było za mało, żeby pracować tak szybko i skutecznie jak doświadczeni programiści z mojego zespołu.

Co takiego wyjątkowego robili pozostali programiści? Poza zaawansowanym programowaniem obiektowym, bardzo sprawnie posługiwali się kolekcjami. Do odpytywania kolekcji korzystali z LINQ podczas gdy ja używałem zwykłych pętli. W naszym projekcie korzystaliśmy z rozwiązania LINQ to SQL więc na co dzień mieliśmy kontakt LINQ. Nowsze aplikacje korzystają na przykład z Entity Framework Core, gdzie w równie dużym stopniu wykorzystuje się LINQ aby odpytywać bazę danych.

Ta książka jest przeznaczona dla Ciebie, jeśli znasz już podstawy języka C# i chcesz sprawnie posługiwać się kolekcjami danych. Lektura sprawi, że opanujesz podstawy LINQ a praca z danymi stanie się przyjemnością.

Ta książka składa się z dwóch rozdziałów. Pierwszy z nich to część teoretyczna. Omawiam w niej metody LINQ, które najczęściej spotykam w projektach. Każdą z metod zilustrowałem odpowiednim przykładem. Drugi rozdział to zadania, które pozwolą Ci przećwiczyć wszystkie metody LINQ przedstawione w pierwszej części. Każde zadanie zawiera pełny kod C#, który można uruchomić. Wystarczy, że go skopiujesz i wkleisz do Visual Studio.

Zapraszam Cię do nauki LINQ bo ta technologia pozwoli Ci pisać lepszy kod C#. Stosując metody LINQ opisane w tej książce zaczniesz zbliżać się do poziomu pracy doświadczonych programistów C#. Nauczysz się tworzyć taki kod, który będzie krótki, bezpieczny i zrozumiały.

Zaczynamy!

Rozdział 1

Teoria LINQ

Co to jest LINQ?

LINQ to część języka C# do pracy z danymi. Nazwa LINQ to akronim od angielskich słów Language Integrated Query. Jest to zestaw metod rozszerzających (ang. extension methods). Te metody ułatwiają odczytywanie informacji z kolekcji takich jak na przykład lista. LINQ świetnie współpracuje na przykład z Entity Framework. Pozwala też odczytywać dane z plików XML.

Wszystko czego nauczysz się z tej książki przyda Ci się do pracy z bazami danych, z plikami XML i nawet do tworzenia testów jednostkowych.

Po co używać LINQ? Po pierwsze znajomość tej technologii jest wymagana przez niemal każdego kto zatrudnia programistów .NET. Po drugie LINQ znacznie uprości Twój kod. Dzięki temu Twoje programy będą krótsze i bardziej czytelne. Po trzecie kiedy zaczniesz korzystać z LINQ to będziesz robił znacznie mniej pomyłek.

Zacznijmy od przykładu. Napišemy program w języku C# na dwa sposoby. Najpierw bez użycia LINQ, a potem z użyciem LINQ. Od razu zauważysz różnicę.

Przykład 1. Jak znaleźć maksymalną wartość - wersja bez użycia LINQ

```
var ages = new List<int>() { 10, 25, 60, 40 };
int theOldest = 0;
foreach (var age in ages)
{
    if (age > theOldest)
    {
        theOldest = age;
    }
}
Console.WriteLine($"Najstarszy ma { theOldest} lat.");
```

Przykład 2. Jak znaleźć maksymalną wartość - wersja z użyciem LINQ

```
var ages = new List<int>() { 10, 25, 60, 40 };  
int theOldest = ages.Max();  
Console.WriteLine($"Najstarszy ma { theOldest} lat.");
```

Wskazówka: Jeśli skorzystasz z metod rozszerzających LINQ takich jak na przykład **Max()** to musisz dodać przestrzeń nazw. Wszystkie omawiane przeze mnie metody wymagają dodania przestrzeni nazw **System.Linq**. Jeżeli o tym zapomnisz to kompilator nie będzie mógł znaleźć metod LINQ. Na szczęście Visual Studio podpowie Ci, że powinieneś dodać przestrzeń. Możesz to zrobić za pomocą kodu:

```
using System.Linq.
```

Teraz kod jest krótki i czytelny.

Spójrz na te dwa fragmenty kodu jeszcze raz.

Zastanów się, który z nich wygląda lepiej.

Której wersji użyjesz w swoich programach?

Co to jest wyrażenie lambda?

Zanim przejdziemy do omówienia najważniejszych metod LINQ, warto poświęcić kilka słów tak zwanym wyrażeniom lambda. Są one wykorzystywane w wielu metodach LINQ. Ale nie tylko. Spotkasz je na przykład w popularnej bibliotece do testowania o nazwie Moq.

Wyrażenie lambda to po prostu funkcja zapisana w uproszczony sposób. Taka funkcja zawsze zawiera specjalny znak `=>`. Spójrzmy na kilka przykładów wyrażenia lambda.

```
// sprawdzenie czy podana liczba jest dodatnia
x => x > 0

// Sprawdzenie czy podany napis nie jest pusty
x => string.IsNullOrEmpty(x)

// Konkatenacja pól: Imie, Nazwisko
x => x.Name + " " + x.Surname

//Mapowanie obiektu jednej klasy na obiekt innej klasy
x => new Employee {Name=x.Name, Surname=x.Surname}
```

Każda z powyższych funkcji składa się z 3 elementów:

1. Lewa część wyrażenia to parametr funkcji. Użyliśmy po prostu `x`. Ale możemy użyć dowolnego innego tekstu.
2. Środkowa część wyrażenia to tzw. strzałka `=>`
3. Prawa część wyrażenia, czyli wartość którą funkcja przyjmie dla podanego argumentu `x`.

Jak widzisz wyrażenie lambda to po prostu funkcja. Przyjmuje parametr i wykonuje pewną operację. W LINQ wyrażenie lambda jest wykonywane na każdym elemencie kolekcji. Za chwilę pokażę Ci kilka przykładów.

Jak zacząć używać LINQ?

Kiedy uczymy się nowej technologii to najtrudniej jest zacząć. Dlatego teraz pokażę Ci jak powinieneś się przygotować do używania LINQ.

Dobra wiadomość jest taka, że nie musisz instalować żadnych pakietów NuGet, ponieważ LINQ jest wbudowany w język C#. Wystarczy, że użyjesz klauzuli **using**. Dzięki temu będziesz mógł użyć każdej metody rozszerzającej LINQ.

Zapamiętaj, że wszystkie metody LINQ są zdefiniowane w przestrzeni nazw **System.Linq**. Wobec tego aby skorzystać z LINQ musimy dodać tylko jedną linię kodu. Ta linia to:

```
using System.Linq;
```

Zakładam, że wiesz jak korzystać z przestrzeni nazw w języku C#. Gdybyś jednak nie był do końca pewien to szybko wyjaśnię. Kiedy otworzysz Visual Studio, to na samej górze pliku zobaczysz kilka liniiek zaczynających się od słowa **using**. Wystarczy, że w tym miejscu dodasz jeszcze jedną taką linijkę.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            ...
        }
    }
}
```

Metoda Select

Pierwsza metoda, którą omówimy to **Select()**. Składnia LINQ jest bardzo podobna do składni języka SQL. Więc jeśli korzystałeś kiedyś z języka SQL to poczujesz się w LINQ jak ryba w wodzie. Metoda Select działa bardzo podobnie do klauzuli SELECT z języka SQL. Pozwala określić transformację jakiej zostanie poddany każdy element oryginalnej kolekcji. Taką transformację nazywamy również projekcją

Założmy, że mamy listę liczb. Liczby te oznaczają wiek naszych pracowników. Chcemy obliczyć ile lat zostało do emerytury każdemu z nich. Zrobmy to za pomocą metody **Select**.

```
List<int> ages = new List<int> {20, 25, 56, 40};  
var yearsToWork = workYears.Select(x => 67-x);
```

Zwróć uwagę na wyrażenie lambda. To tutaj skupia się cała logika naszego programu. Zmienna x reprezentuje tutaj liczbę z kolekcji wejściowej. Zamiast tej liczby w kolekcji wyjściowej pojawi się wynik odejmowania: 67 - x.

Metoda Select działa tak:

- Na wejściu naszego programu jest oryginalna lista
- Na podstawie tej listy tworzymy drugą listę.
- Druga lista zawsze ma tyle samo elementów co lista oryginalna.
- Druga lista jest wynikiem działania metody Select

Zatem w naszym przykładzie lista `yearsToWork` będzie miała również 4 elementy. Spróbuj uruchomić powyższy fragment. Wystarczy, że umieścisz go w metodzie `Main`. Pamiętaj tylko o dodaniu odpowiednich przestrzeni nazw.

W Polsce obowiązuje wiek emerytalny 67 lat dla mężczyzn i 65 lat dla kobiet. W naszym przykładzie przyjęliśmy wiek 67 lat dla wszystkich. Zrobiliśmy to dla uproszczenia kodu.

Metoda Where

W poprzednim przykładzie lista wyjściowa zawsze miała tyle samo elementów co lista wejściowa. Ale czasami chcemy wydobyć ze zbioru tylko niektóre jego elementy. Taką operację nazywamy filtrowaniem. W LINQ do filtrowania służy metoda `Where()`. Ta metoda działa tak samo jak klauzula `WHERE` w języku `SQL`.

Przykład. Chcemy znaleźć tylko pracowników starszych niż 30 lat. W tej sytuacji zastosujemy metodę `Where`.

```
var ages = new List<int> { 10, 25, 60, 40 };  
var seniorsAges = ages.Where(x => x > 30);
```

Argumentem metody `Where` jest wyrażenie lambda: `x => x > 30`. Oznacza ono warunek logiczny. Ten warunek jest spełniony tylko dla dwóch elementów listy. Wniosek jest prosty: na naszej liście wyjściowej `yearsToRetirement` pojawią się tylko dwa elementy. Pozostałe elementy, które nie spełniają warunku logicznego zostaną pominięte.

Później bardzo łatwo możemy wyświetlić rezultat. Zmienna `seniorsAges` jest typu `IEnumerable<int>`. Dzięki temu możemy odczytać jej elementy za pomocą pętli `foreach`, a następnie wyświetlić je na ekranie.

```
foreach (var age in seniorsAges)  
{  
    Console.WriteLine(age);  
}
```

Metoda First

Każdy z nas w jakiś sposób planuje swoją pracę. Ja na przykład biorę kartkę papieru i zapisuję na niej wszystkie zadania. Jeśli te zadania są tak samo ważne to bez znaczenia jest to, od którego z nich zacznę. W takiej sytuacji najprościej jest wykonać pierwsze zadanie z listy. Potem powtarzamy tę czynność, aż do momentu gdy nasza lista będzie pusta.

Aby zrobić to samo w języku C# to najlepiej jest użyć LINQ. Wykorzystamy jego metodę **First()**. Ta metoda jest przydatna kiedy chcemy pobrać tylko pierwszy element kolekcji.

Przykład: Wyświetlenie pierwszego elementu kolekcji

```
var tasks = new List<string>
{
    "Odpowiedzieć na maile",
    "Napisać trochę kodu",
    "Zrobić code review",
    "Odtworzyć błąd zgłoszony przez testera"
};

string task = tasks.First();
Console.WriteLine(task);
```

Metoda **First()** ma tylko jedną wadę. Jeśli kolekcja okaże się pusta, to zostanie zgłoszony wyjątek `System.InvalidOperationException`. Pojawi się komunikat: "sequence contains no elements". Nasz program przestanie działać. Dokładnie to się stanie kiedy uruchomisz następujący kod:

```
var tasks = new List<string>();
string task = tasks.First();
```

To częsty problem. Zaraz pokażę Ci jak możesz go rozwiązać.

Metoda FirstOrDefault

Wiemy już, że metoda **First** zgłasza wyjątek kiedy lista jest pusta. Metoda **FirstOrDefault** jest pod tym względem lepsza bo nie zgłasza wyjątku. Zamiast tego przyjmuje ona wartość domyślną dla danego typu danych. Stąd słowo Default w nazwie metody.

```
var tasks = new List<string>();
string task = tasks.FirstOrDefault();
if(task == null)
{
    Console.WriteLine("Koniec pracy! Brawo!");
}
```

Zmienna `tasks` przechowuje listę typu `string`. `String` jest typem referencyjnym, a wszystkie typy referencyjne mają wartość domyślną `null`. Dlatego w tym przykładzie porównaliśmy zmienną `task` do `null`. Dzięki temu jesteśmy w stanie odpowiednio zareagować kiedy lista jest pusta.

Czy używając metody **First()** możemy zapobiec awarii programu? Tak. Możemy to zrobić za pomocą instrukcji `try-catch`, ale kod będzie wyglądał dość dziwnie:

```
var tasks = new List<string>();
try
{
    string task = tasks.First();
}
catch(InvalidOperationException)
{
    Console.WriteLine("Koniec pracy! Brawo!");
}
```

Poza tym ten kod jest mało precyzyjny bo jaką mamy pewność, że przyczyną wyjątku była pusta lista?

Podsumowując, jeśli lista może być pusta to użyj metody **FirstOrDefault**.

Metoda Single

Kolejna metoda, którą musisz poznać to **Single**. Pozwól, że wyjaśnię jej działanie na przykładzie. Załóżmy, że rozwijasz aplikację dla firmy z branży energii odnawialnej. W bazie danych znajdują się informacje o turbinach wiatrowych. Uproszczony model turbiny wygląda tak:

```
public class Turbine
{
    public string Name { get; set; }
    public int Diameter { get; set; }
}
```

Jak widzisz każda turbina posiada swoją nazwę oraz średnicę. Lista turbin zostanie pobrana z bazy danych, ale dla uproszczenia użyjemy zwykłej listy.

```
var turbines = new List<Turbine>()
{
    new Turbine { Name="V164-9.5 MW", Diameter=164 },
    new Turbine { Name="V155-3.6 MW", Diameter=155 },
    new Turbine {Name="V117-4.2 MW", Diameter=117 }
};
```

Nasze zadanie jest takie: Kiedy użytkownik poda nazwę turbiny to musimy pokazać jej średnicę. Przyjmijmy, że nazwa turbiny jest unikalna, bo tylko wtedy możemy użyć metody **Single**.

```
var turbine = turbines
    .Single(turbine => turbine.Name == "V164-9.5 MW");
Console.WriteLine(turbine.Diameter);
```

Program pobierze element, który spełnia podany warunek.

Jednak gdyby okazało się, że nazwy nie są już unikalne, to nasz program przestanie działać prawidłowo. Załóżmy, że w naszym programie występują dwie turbiny

o tej samej nazwie. W takiej sytuacji metoda **Single** zwróci wyjątek. Zatem dodajmy do naszej kolekcji jeden dodatkowy element. W rezultacie nazwy nie są już unikalne. Spróbuj uruchomić ten program:

```
var turbines = new List<Turbine>()
{
    new Turbine {Name="V164-9.5 MW", Diameter=164 },
    new Turbine {Name="V155-3.6 MW", Diameter=155 },
    new Turbine {Name="V117-4.2 MW", Diameter=117 },
    new Turbine {Name="V117-4.2 MW", Diameter=118 }
};

var turbine = turbines
    .Single(turbine => turbine.Name == "V117-4.2 MW");
Console.WriteLine(turbine.Diameter);
```

Kiedy uruchomisz ten kod to zauważysz, że wynik nie wyświetli się na ekranie. Metoda `Console.WriteLine` nie zostanie nawet uruchomiona. Bo zanim to nastąpi, zostanie zgłoszony wyjątek: `System.InvalidOperationException: Sequence contains more than one matching element`.

Ten błąd oznacza, na naszej liście znajdują się dwa elementy spełniające podany warunek. Metoda **Single** oczekuje jednak, że będzie tylko jeden element. Kiedy znajdzie ona więcej niż jeden element to nie wie, o który z dwóch elementów chodzi. A jeśli program nie wie jak się zachować to zgłasza wyjątek.

A co się stanie jeżeli żaden element listy nie spełnia określonego warunku? Wtedy program również nie może ustalić o który element chodzi. Pojawi się taki sam wyjątek, ale z inną wiadomością wyjaśniającą przyczynę błędu:

`System.InvalidOperationException: „Sequence contains no matching element”`

W takich sytuacjach przydatna okazuje się metoda **SingleOrDefault**, która zamiast zgłaszania wyjątku, zwróci null. Tak samo działa metoda **FirstOrDefault**.

Metoda Distinct

Założmy, że jesteś nauczycielem w szkole. Przeprowadziłeś sprawdzian. Uczniowie rozwiązyali zadania. Oceniłeś je i masz wyniki. Teraz chcesz omówić je z uczniami. Masz pomysł, żeby podczas omówienia sprawdzianu pokazać jakie pojawiły się oceny. Masz listę trzydziestu ocen i wiele z nich się powtarza. W języku C# bardzo łatwo możemy znaleźć unikalne wartości. Wystarczy użyć metody LINQ o nazwie **Distinct**.

Distinct to odpowiednik klauzuli **SELECT DISTINCT** z języka SQL. Dzięki tej metodzie możemy usunąć duplikaty z oryginalnej listy.

```
var marks = new List<int> {1,2,2,2,2,4,4,4,5,5,6};  
var uniqueMarks = marks.Distinct();
```

```
Console.WriteLine(string.Join(", ", uniqueMarks));
```

Nasz wynik będzie zawierał tylko unikalne wartości. Na przykład liczba 2 pojawi się w naszym wyniku tylko jeden raz. Mimo, że w oryginalnej kolekcji występuje ona aż 4 razy.

Metoda **Distinct** przyda Ci się zawsze gdy chcesz dowiedzieć się ile unikalnych wartości znajduje się w określonej kolekcji.

Metoda Any

Dzięki tej metodzie **Any** możemy sprawdzić czy lista zawiera jakiegokolwiek elementy. Warto użyć tej metody zamiast `marks.Count != 0`.

```
var marks = new List<int> { 2,4,5,5,6 };
bool marksIsNotEmpty = marks.Any();

if (marksIsNotEmpty)
{
    Console.WriteLine("Są już oceny.");
}
else
{
    Console.WriteLine("Jeszcze nie ma żadnych ocen");
}
```

Możesz łatwo sprawdzić czy którykolwiek z uczniów otrzymał ocenę celującą ponieważ metoda **Any** występuje również w wersji z wyrażeniem lambda. Możesz określić dowolny warunek. W poniższym przykładzie sprawdzamy czy na liście istnieje co najmniej jedna ocena równa 6.

```
var marks = new List<int> { 2, 4, 5, 5, 6 };
bool anyHighestMark = marks.Any(mark => mark == 6);

if (anyHighestMark)
{
    Console.WriteLine("Jest ocena maksymalna.");
}
else
{
    Console.WriteLine("Niema oceny maksymalnej.");
}
```

Metoda All

Metoda **All** pozwala sprawdzić czy wszystkie elementy listy spełniają określony warunek. Na przykład możemy sprawdzić czy wszyscy uczniowie zdali egzamin. Robimy to za pomocą wyrażenia lambda. Za jego pomocą określamy warunek mówiący, że wszystkie liczby na liście muszą być większe niż 1.

```
var marks = new List<int> { 1,2,2,2,2,4,4,4,5,5,6 };
bool allStudentsPassed = marks.All(x => x > 1);

if (allStudentsPassed)
{
    Console.WriteLine("Wszyscy zdali egzamin");
}
else
{
    Console.WriteLine("Niektórzy nie zdali.");
}
```

Metoda **All** zwraca wynik typu bool. Dzięki temu możemy podjąć decyzję co zrobić dalej z tym wynikiem. W naszym przykładzie po prostu wyświetlamy jeden napis, albo drugi.

Metoda OrderBy

Jeśli chcesz posortować tablicę, to skorzystaj z metody **OrderBy**. Sortowanie liczb jest trywialne. Nieco trudniej jest posortować obiekty. Bo musimy wskazać pole klasy, które będzie wykorzystane do sortowania.

Przykład. Bieg na 100 metrów. Mamy listę biegaczy wraz z ich wynikami. Zawodnicy są umieszczeni na liście w losowej kolejności (wyniki są prawdziwe i pochodzą z Letnich Igrzysk Olimpijskich 2021). Naszym zadaniem jest posortować listę w taki sposób, żeby najlepszy wynik (czyli najkrótszy czas) był na początku a najgorszy na końcu. Możemy to zrobić tak:

```
var runners = new List<Runner>()
{
    new Runner {
        Name = "Andre De Grasse",
        Time = TimeSpan.Parse("00:00:08.93")
    },
    new Runner {
        Name = "Lamont Marcell Jacobs",
        Time = TimeSpan.Parse("00:00:08.84")
    },
    new Runner
    {
        Name = "Fred Kerley",
        Time = TimeSpan.Parse("00:00:08.89")
    },
};

var orderedRunners = runners
    .OrderBy(r => r.Time);

foreach(var runner in orderedRunners)
    Console.WriteLine(runner.Name);
```

Widać, że metody LINQ możemy wykorzystać kiedy lista jest zawiera liczby, napisy, a także bardziej skomplikowane obiekty takie jak na przykład Runner.

W wyrażeniu lambda możemy użyć dowolnej litery. W tym przypadku użyliśmy `r`, ale moglibyśmy użyć na przykład słowa `runner`. Często w kodzie spotykamy się z łańcuchem kilku metod LINQ. Wtedy w każdym wyrażeniu lambda możemy użyć innej nazwy:

```
var orderedRunners = runners
    .OrderBy(runner => runner.Time)
    .Select(r => r.Name);
```

Choć takie rozwiązanie jest również poprawne:

```
var orderedRunners = runners
    .OrderBy(runner => runner.Time)
    .Select(runner => runner.Name);
```

Jeden z moich znajomych programistów kiedyś podzielił się ze mną swoim sposobem. Powiedzia tak: *"W wyrażeniach lambda zawsze nazywam zmienną X"*.

```
var orderedRunners = runners
    .OrderBy(x => x.Time)
    .Select(x => x.Name);
```

Wybór konwencji należy do Ciebie. W moich własnych projektach lubię używać pełnych nazw. Na przykład `runner`. Jednak oprogramowanie zwykle piszemy w zespole. Co więcej często pracujemy w projektach, które nie są nowe. Wtedy najlepiej będzie jeśli zastosujesz takie samo podejście jak Twój poprzednicy. Wystarczy zajrzeć do kodu i sprawdzić. Potem powinieneś dostosować się do istniejącej konwencji.

WSKAZÓWKA. Pamiętaj o spójności stylu w ramach projektu. Dzięki temu cały kod programu będzie wyglądał podobnie, niezależnie od tego czy pisała go jedna osoba czy dziesięć osób. Słyszałem kiedyś opinię, że idealny kod wygląda tak jakby był napisany przez jedną osobę. W pełni się z nią zgadzam. Właśnie w ten sposób piszę programy.

Metoda OrderByDescending

Ta metoda działa tak samo jak `OrderBy`, ale sortuje wyniki w odwrotnej kolejności. Metody `OrderByDescending` możesz użyć aby umieścić najmniejszą wartość na początku listy a największą na końcu.

Warto wykorzystać tę metodę kiedy chcesz znaleźć najnowsze wpisy w bazie danych. Zrobimy to bez żadnego problemu, kiedy każdy rekord w bazie danych posiada datę utworzenia.

Przykład. Chcemy wyświetlić zawodników w określonej kolejności. Na początku pojawi się zawodnik z najgorszym, czyli najdłuższym czasem. Natomiast na końcu pojawi się zawodnik z najlepszym, czyli najkrótszym czasem.

```
var orderedRunners = runners
    .OrderByDescending(runner => runner.Time)
    .Select(r => r.Name);
```

Rozdział 2

Zadania LINQ

O zadaniach

Muszę Ci pogratulować. Właśnie dotarłeś do drugiej części książki i liczę, że na tym nie poprzestaniesz. Bo to dopiero połowa sukcesu. Przed nami zdania, które pozwolą Ci się sprawdzić. Takie same zadania pojawią się w Twojej pracy, albo na rozmowach o pracę.

W pierwszym rozdziale przedstawiłem 11 najważniejszych metod LINQ. Omawialiśmy je w takiej kolejności:

- Select
- Where
- First
- FirstOrDefault
- Single
- Distinct
- Any
- All
- OrderBy
- OrderByDescending
- Max

Zadania z tej części książki będą prezentowane w tej samej kolejności. Dla każdej metody LINQ otrzymasz co najmniej jedno zadanie. Wykonaj je wszystkie. Pamiętaj, że jesteś w stanie je rozwiązać posiłkując się poprzednią częścią podręcznika. Więc w razie kłopotów możesz spokojnie cofnąć się do pierwszego rozdziału i rozwiązać wszelkie swoje wątpliwości. Gdyby to się nie udało to napisz do mnie. Mój adres mailowy znajdziesz na końcu książki.

Teraz przyszedł czas próby. Sprawdźmy ile zapamiętałeś z rozdziału teoretycznego. Zobaczymy czy potrafisz rozwiązać codzienne zadania programisty C#. To od Twojego wysiłku w tej chwili będzie zależała Twoja skuteczność w przyszłej pracy. Dlatego wykonuj skrupulatnie wszystkie zadania. Twój wysiłek na pewno będzie procentował w przyszłości.

Metoda Select - zadanie 1

Założmy, że mamy listę nazwisk. Uzupełnij wyrażenie lambda w taki sposób aby program wyświetlił dokładnie 5 pierwszych znaków imienia i nazwiska. Na przykład zamiast "Dariusz Kacban" program powinien wyświetlić "Dariu".

```
var names = new List<string>();
names.Add("Jan Nowak");
names.Add("Adam Kowalski");
names.Add("Dariusz Kacban");

var namesResult = names.Select(x => ...);
foreach(var name in namesResult)
{
    Console.WriteLine(name);
}
```

Wskazówka: Jak pobrać 7 pierwszych znaków dowolnego napisu? Gdybyśmy operowali na pojedynczej zmiennej typu String to moglibyśmy użyć metody Substring w taki sposób:

```
string text = "Bardzo długi tekst";
string shortText = text.Substring(0,7);
```

To oczywiście przykład dla pojedynczej zmiennej typu string.

Ale operując na kolekcji zmiennych typu string również możemy użyć metody Substring. Wystarczy umieścić ją w wyrażeniu lambda.

Metoda Select - zadanie 2

Na farmie wiatrowej znajdują się turbiny wiatrowe. Każda z nich posiada trzy łopaty. To ta część turbiny, która obraca się podczas pracy. Pod wpływem siły wiatru łopaty mogą się uginać. W zależności od kierunku wiatru wyginają się albo do przodu albo do tyłu.

System odczytujący dane o wychyleniu łopat zapisuje dane w taki sposób:

- liczby przedziału (0,12] - oznaczają wychylenie do tyłu
- liczba 0 oznacza położenie neutralne np. przy bezwietrznej pogodzie
- liczby ujemne z przedziału (0, -12] - oznaczają wychylenie do przodu, np. kiedy wiatr nagle zmieni kierunek i zacznie wiać od tyłu.

Twoje zadanie będzie polegało na tym, żeby wśród danych pomiarowych znaleźć największą wartość. Bez względu na kierunek wiatru. A więc musisz odczytać wartość bezwzględną z każdej liczby. Uzupełnij wyrażenie lambda:

```
var yaw = new List<double>();  
yaw.Add(0);  
yaw.Add(2.5);  
yaw.Add(2.8);  
yaw.Add(5.1);  
yaw.Add(0.4);  
yaw.Add(0.1);  
yaw.Add(-0.3);  
yaw.Add(-4.8);  
yaw.Add(-6.8);  
  
var maxYaw = yaw.Max(x => ...);  
Console.WriteLine(maxYaw);
```

Wskazówka: Dla pojedynczej zmiennej wartość bezwzględną odczytujemy tak:

```
int x = -5;  
Math.Abs(x);
```

Metoda Select - zadanie 3

Czy wiesz, że prawie jedna czwarta większości Twoich zakupów to podatek? Na przykład kiedy kupujesz telewizor za 1230 PLN to sprzedawca otrzymuje tylko 1000 PLN, a pozostałe 230 PLN trafia do Urzędu Skarbowego jako podatek VAT.

A więc w naszym przykładzie ostateczna cena składa się z dwóch składników:
 $1230 \text{ PLN} = 1000 \text{ PLN} + 230 \text{ PLN}$

Można z tego utworzyć ogólny wzór:

$\text{CENA} = \text{kwota netto} + \text{podatek VAT}$

$\text{CENA} = \text{kwota netto} + (\text{kwota netto}) * 23\%$

CENA to cena ostateczna. Nazywamy ją ceną brutto. To jest ta kwota, którą klient faktycznie płaci.

Po tym podatkowym wstępie przejdźmy do zadania. Wcielimy się w rolę programisty, który tworzy aplikację dla sieci sklepów TESCO. Otrzymujemy od TESCO listę produktów oraz zlecenie. Zlecenie polega na zbudowaniu programu do drukowania etykiet z cenami produktów. Na etykietach ma widnieć cena brutto. Jednak ceny są podane w postaci netto. Musimy obliczyć ceny brutto.

Twoje zadanie: Zmodyfikuj wyrażenie lambda w metodzie Select tak, aby wyświetlić ceny brutto. Zmodyfikuj tylko linijkę oznaczoną komentarzem **TODO**, a konkretnie samo wyrażenie lambda. Docelowo program powinien wyświetlić taki tekst:

1,23 zł

2,46 zł

0,25 zł

3,01 zł

Wskazówka: każda cena powinna mieć dwa miejsca po przecinku. Do zaokrąglenia liczb wykorzystaj więc metodę `Math.Round`.

```
public class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
    public Product(string name, decimal price)
    {
        Name = name;
        Price = price;
    }
}

static void Main(string[] args)
{
    var products = new List<Product>();
    products.Add(new Product("woda żywiec 0.5L", 1.00m));
    products.Add(new Product("woda Nałęczowianka 1L", 2.00m));
    products.Add(new Product("bułka pszenna 50g", 0.20m));
    products.Add(new Product("chleb krojony 600g", 2.45m));

    var productsGrossValues = products
        .Select(x => ... ); //TODO - tu zmodyfikuj wyrażenie lambda

    foreach(var grossPrice in productsGrossValues)
    {
        Console.WriteLine(grossPrice);
    }
}
```

Metoda Where - zadanie 1

Pierwsze zadanie z filtrowania za pomocą metody **Where** zaczniemy od bardzo popularnego przypadku. Stworzymy wyszukiwarkę. Taka wyszukiwarka jest częścią niemal każdej webowej aplikacji. Wyobraźmy sobie, że użytkownicy chcą zobaczyć np. listę produktów dostępnych w sklepie. Użyj metody `Contains` aby na liście pozostały tylko te elementy, które zawierają tekst mieszczący się w zmiennej `searchFilter`.

```
public class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
    public Product(string name, decimal price)
    {
        Name = name;
        Price = price;
    }
}

string searchFilter = "kro";
var products = new List<Product>();
products.Add(new Product("woda żywiec 0.5L", 1.00m));
products.Add(new Product("woda Nałęczowianka 1L", 2.00m));
products.Add(new Product("bułka pszenna 50g", 0.20m));
products.Add(new Product("chleb krojony 600g", 2.45m));

var productsFiltered = products
    .Where(x => ...) //TODO tutaj zmodyfikuj wyrażenie lambda
    .Select(x => x.Name);

foreach(var product in productsFiltered)
{
    Console.WriteLine(product);
}
```

Metoda Where - zadanie 2

Kolejnym bardzo często używanym przeze mnie rodzajem filtrowania jest filtrowanie po dacie. Robimy to kiedy na przykład chcemy pobrać z bazy danych tylko te produkty, które nie są przeterminowane.

Twoje zadanie będzie polegało na wyświetleniu tylko przeterminowanych produktów. Dzięki Twojej pracy pracownicy sklepu będą mogli łatwiej zlokalizować takie produkty i zabrać je ze sklepowych półek.

Jeżeli prawidłowo rozwiążesz zadanie to zobaczysz taki wynik:

bułka pszenna 50g

chleb krojony 600g

Wskazówka:

- Użyj właściwości `DateTime.UtcNow` aby ustalić aktualną datę
- Użyj operatora logicznego OR, czyli: `||` aby uwzględnić również wartość null
- Zmodyfikuj tylko wskazane wyrażenie lambda.

```
public class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
    public DateTime? ValidTo { get; set; }
}

public Product(string name, decimal price, DateTime? validTo)
{
    Name = name;
    Price = price;
    ValidTo = validTo;
}
```

```
static void Main(string[] args)
{
    var products = new List<Product>()
    {
        new Product("woda żywiec 0.5L", 1.00m, DateTime.Parse("10-05-2022")),
        new Product("woda Nał. 1L", 2.00m, DateTime.Parse("18-01-2022")),
        new Product("bułka pszenna 50g", 0.20m, DateTime.Parse("20-09-2021")),
        new Product("chleb krojony 600g", 2.45m, null)
    };

    var expiredProducts = products
        .Where(x => x.ValidTo < DateTime.UtcNow || x.ValidTo==null)
        .Select(x => x.Name);

    foreach(var product in expiredProducts)
    {
        Console.WriteLine(product);
    }

    var expiredProducts = products
        .Where(x => ...) //TODO - uzupełnij wyrażenie lambda
        .Select(x => x.Name);

    foreach(var product in expiredProducts)
    {
        Console.WriteLine(product);
    }
}
```

Metoda First - zadanie

W tym zadaniu zobaczysz coś, co spotkasz w życiu. Załóżmy, że w projekcie, nad którym pracujesz istnieje pewna metoda zwracająca listę produktów w sklepie. Sprawdziłeś schemat bazy danych. Nazwy produktów muszą być unikalne. Zatem na pewno nie wystąpią dwa produkty o tej samej nazwie. Wynika z tego, że ta metoda zwróci zawsze tylko jeden produkt. Dzięki temu możemy wywołać metodę **First** bezpośrednio na tej metodzie.

W podanej niżej formie program nawet się nie skompiluje. Zmodyfikuj go tak, aby się kompilował oraz wyświetlał nazwę szukanego produktu i jego cenę.

Podpowiedź: Do rozwiązania wystarczy dodać w odpowiednim miejscu kod: **.First()**. Znajdź to miejsce i dodaj ten kod, a program zadziała.

Oczekiwany rezultat:

nazwa: woda żywiec 0.5L, cena 1,00

```
public class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
    public DateTime? ValidTo { get; set; }

    public Product(string name, decimal price, DateTime? validTo)
    {
        Name = name;
        Price = price;
        ValidTo = validTo;
    }
}
```

```
public class ProductRepository
{
    List<Product> products = new List<Product>()
    {
        new Product("woda żywiec 0.5L", 1.00m, DateTime.Parse("10-05-2022")),
        new Product("woda Nałęczowianka 1L", 2.00m, DateTime.Parse("18-01-2022")),
        new Product("bułka pszenna 50g", 0.20m, DateTime.Parse("20-09-2021")),
        new Product("chleb krojony 600g", 2.45m, null)
    };

    public IEnumerable<Product> FindProducts(string name)
    {
        return products.Where(product => product.Name == name);
    }
}

class Program
{
    static void Main(string[] args)
    {
        var repository = new ProductRepository();
        var product = repository.FindProducts("woda żywiec 0.5L");

        Console.WriteLine($"nazwa: {product.Name}, cena {product.Price}");
    }
}
```

Metoda FirstOrDefault - zadanie

Czasami dopuszczamy sytuację, że nie znajdziemy szukanej informacji w bazie danych (albo w dowolnym innym źródle). Metoda First() nie zadziała. Program zgłosi wyjątek i przestanie działać. Niemal na pewno spotkasz się z tym błędem w swojej pracy. Co wtedy? Z pierwszej części książki wiesz już, że powinniśmy dodać odpowiednie zabezpieczenie. Za chwilę to przećwiczymy.

Twoje zadanie ma dwa etapy:

1. Ustal dlaczego podany program nie działa. Napraw go stosując wytyczne z pierwszej części książki (rozdział na temat metody FirstOrDefault).
2. Jeżeli na liście nie będzie szukanego produktu to wyświetlimy informację: "Przepraszamy, podany produkt nie istnieje w bazie danych". Użyj porównania do wartości null.

```
public class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
    public DateTime? ValidTo { get; set; }

    public Product(string name, decimal price, DateTime? validTo)
    {
        Name = name;
        Price = price;
        ValidTo = validTo;
    }
}
```

```
public class ProductRepository
{
    List<Product> products = new List<Product>()
    {
        new Product("woda żywiec 0.5L", 1.00m, DateTime.Parse("10-05-2022")),
        new Product("woda Nałęczowianka 1L", 2.00m, DateTime.Parse("18-01-2022")),
        new Product("bułka pszenna 50g", 0.20m, DateTime.Parse("20-09-2021")),
        new Product("chleb krojony 600g", 2.45m, null)
    };

    public IEnumerable<Product> FindProducts(string name)
    {
        return products.Where(product => product.Name == name);
    }
}

class Program
{
    static void Main(string[] args)
    {
        var repository = new ProductRepository();
        var product = repository.FindProducts("aaaaa").First();

        Console.WriteLine($"nazwa: {product.Name}, cena {product.Price}");
    }
}
```

Metody Single i SingleOrDefault - zadania

Teraz spójrz na poniższy przykład kodu. Program oczywiście nie zadziała poprawnie. Poprzedni programista nieświadomie popełnił pewien błąd. Twoje zadanie będzie polegało na naprawieniu programu.

Zadania dla Ciebie:

1. Spójrz na kod i powiedz dlaczego nie działa on prawidłowo. Znajdź błąd.
2. Wykorzystaj metody Where oraz First żeby poprawić kod.
3. Skróć kod - zamiast Where i First użyj metody Single z wyrażeniem lambda.
4. Przetestuj czy wyszukiwanie produktów działa prawidłowo. Zamiast "pszenny" użyj słów: "żywiec", "500g" oraz "krojony".
5. Użyj słowa "woda". Czy teraz program działa? Nie powinien. Wyjaśnij dlaczego program nie działa i jak można to naprawić.
6. Użyj odpowiedniej metody aby program działał prawidłowo.

```
public class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
    public DateTime? ValidTo { get; set; }

    public Product(string name, decimal price, DateTime? validTo)
    {
        Name = name;
        Price = price;
        ValidTo = validTo;
    }
}
```

```
public class ProductRepository
{
    List<Product> products = new List<Product>()
    {
        new Product("woda żywiec 0.5L", 1.00m, DateTime.Parse("10-05-2022")),
        new Product("woda Nałęczowianka 1L", 2.00m, DateTime.Par-
se("18-01-2022")),
        new Product("chleb pszenny 500g", 3.60m, DateTime.Parse("20-09-2021")),
        new Product("chleb krojony 600g", 4.45m, null)
    };

    public Product FindProduct(string name)
    {
        return products.First();
    }
}

class Program
{
    static void Main(string[] args)
    {
        var repository = new ProductRepository();
        var product = repository.FindProduct("pszenny");
        Console.WriteLine($"nazwa: {product.Name}, cena {product.Price}");
    }
}
```

Metoda Distinct - zadanie

W naszym systemie sklepowym przechowujemy informacje o oferowanych produktach. Każdy egzemplarz, który oferujemy będzie posiadał osobny wpis w bazie danych. Więc jeżeli na półce stoi 100 butelek wody żywiec, to w bazie danych będzie istnieć 100 rekordów. Każdy z nich posiada taką samą nazwę i cenę.

Kiedy wejdiesz do dowolnego sklepu to na półce z towarem znajdziesz również etykietę. Etykieta pokazuje nazwę i cenę produktu. Naszym zadaniem będzie stworzenie takich etykiet. Jednak czy musimy drukować ich aż 100? Oczywiście, że nie. Wystarczy nam jedna.

Spójrz więc na poniższy kod i zastanów się:

- Ile razy wyświetli się nazwa tego samego produktu?
- Co należy zrobić, żeby nazwa wyświetliła się tylko jeden raz?
- Użyj metody **Distinct** aby wyświetlić nazwy produktów bez duplikatów.

```
static void Main(string[] args)
{
    List<Product> products = new List<Product>()
    {
        new Product("woda żywiec 0.5L", 1.00m, DateTime.Parse("10-05-2022")),
        new Product("woda żywiec 0.5L", 1.00m, DateTime.Parse("10-05-2022")),
        new Product("woda żywiec 0.5L", 1.00m, DateTime.Parse("10-08-2022")),
        new Product("woda żywiec 0.5L", 1.00m, DateTime.Parse("10-05-2023"))
    };

    foreach(var product in products)
    {
        Console.WriteLine(product.Name);
    }
}
```

Metoda Any - zadanie

Czasami zdarzają się w sklepie towary, które nie cieszą się zbyt dużym zainteresowaniem klientów. Taki towar zajmuje miejsce na półkach i w magazynach. Poza tym po jakimś czasie towar staje się przeterminowany. Od razu widać, że takie towary generują straty dla właściciela sklepu. Dlatego powinniśmy je usunąć z magazynu. Trzeba jednak zacząć od ich wyszukania.

Twoje zadanie: Zmodyfikuj tylko metodę `IsSold` w taki sposób, aby sprawdzała czy produkt o podanej nazwie został sprzedany przynajmniej jeden raz.

Program powinien wyświetlić następujące wartości:

False

False

False

True

Oto kod programu, który będziesz modyfikował:

```
public class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
    public DateTime? ValidTo { get; set; }
    public bool Sold { get; set; }

    public Product(string name, decimal price, DateTime? validTo, bool sold
= false)
    {
        Name = name;
        Price = price;
        ValidTo = validTo;
        Sold = sold;
    }
}
```

```
public class ProductRepository
{
    List<Product> products = new List<Product>()
    {
        new Product("woda żywiec 0.5L", 1.00m, DateTime.Parse("10-05-2022")),
        new Product("woda Nałęczowianka 1L", 2.00m, DateTime.Parse("18-01-2022")),
        new Product("bułka pszenna 50g", 0.20m, DateTime.Parse("20-09-2021")),
        new Product("chleb krojony 600g", 2.45m, null, true)
    };

    public Product FindProduct(string name)
    {
        return products.First();
    }

    public bool IsSold(string name)
    {
        return products.Any(x => true);
    }
}

class Program
{
    static void Main(string[] args)
    {
        ProductRepository productsRepository = new ProductRepository();
        Console.WriteLine(productsRepository.IsSold("woda żywiec 0.5L"));
        Console.WriteLine(productsRepository.IsSold("woda Nałęczowianka 1L"));
        Console.WriteLine(productsRepository.IsSold("bułka pszenna 50g"));
        Console.WriteLine(productsRepository.IsSold("chleb krojony 600g"));
    }
}
```

Uwaga: W podanym przykładzie zastosowałem pewne uproszczenie. Klasa `ProductRepository` to przykład klasy repozytorium. Repozytorium to bardzo popularny wzorzec projektowy. Ogólnie chodzi w nim o to, że klasa repozytorium posiada pewne metody umożliwiające dostęp do danych (odczytuje dane np. z bazy danych SQL albo z pliku CSV). Tworząc taką klasę skupiamy się tylko na dostępie do danych. Dlatego w takiej klasie powinny być jedynie metody pobierające dane ze źródła, albo modyfikujące je. Natomiast nie powinniśmy umieszczać w repozytorium bardziej skomplikowanej logiki.

Na przykład niektóre metody służą jedynie do pobrania wszystkich produktów z bazy danych, albo do pobrania jednego produktu. Nie ma w nich tak zwanej logiki biznesowej.

Dlatego takie metody mogłyby być częścią klasy `ProductRepository`:

- `GetAll()`
- `Get(int id)`
- `Add(Product product)`
- `Update(Product product)`
- `Delete(int id)`

Ale istnieją także metody, które mają w sobie logikę biznesową. Na przykład sprawdzamy w nich czy dany produkt został sprzedany, albo generujemy fakturę na podstawie danego produktu. Zadanie tych metod wykracza już poza dostęp do danych.

Dlatego takie metody nie powinny znaleźć się w repozytorium:

- `IsSold(string name)`
- `GetProductAndCreateInvoice(int id)`
- `GetProductWithVatTax(int id)`
- `AddProductToInvoice(int id, Invoice invoice)`

Zatem gdzie jest miejsce takich metod? W warstwie domenowej aplikacji, czyli na przykład w serwisach domenowych. Pamiętaj, że to co widzisz w zadaniu to jedynie uproszczenie.

Metoda All - zadanie

Metoda **All** pozwala nam sprawdzić, czy wszystkie wyszukane przez nas elementy kolekcji spełniają określony warunek. Pamiętaj, że metody LINQ takie jak **Where**, **Select**, **Union**, **Take** albo **Join** zwracają typ danych IEnumerable. Dzięki temu możemy wywołać kilka metod na jednej zmiennej. Taki styl programowania nazywa się Fluent Interface albo Method Chaining.

Przykład:

```
products.Where(x => x.Sold == false).OrderBy(x => x.Price).Select(x =>
x.Name);
```

To samo możemy zapisać w bardziej czytelny sposób:

```
products
    .Where(x => x.Sold == false)
    .OrderBy(x => x.Price)
    .Select(x => x.Name);
```

Zadanie:

Program, który tworzysz służy do drukowania etykiet w sklepach. Twój klient informuje Cię o nowym ograniczeniu. Nazwy produktów na etykietach mogą mieć maksymalnie 18 znaków. Kiedy nazwa jest dłuższa to musimy to wykryć i odpowiednio zareagować. Moglibyśmy na przykład skrócić nazwę, ale to zadanie na później. Teraz skupimy się tylko na wykryciu takiej sytuacji.

Cała siła metod LINQ polega na użyciu wyrażeń lambda. Tak samo jest w tym zadaniu. Wystarczy, że zmienisz wyrażenie lambda w wywołaniu metody **All**. Wykorzystaj zmienną `lengthLimit`.

Uwaga: Zauważ, że w programie pojawiła się z kolejną metodą LINQ o nazwie **Count**. Służy ona do obliczenia ile elementów zawiera dana kolekcja.

```
static void Main(string[] args)
{
    List<Product> products = new List<Product>()
    {
        new Product("woda żywiec 0.5L", 1.00m, DateTime.Parse("10-05-2022")),
        new Product("woda Nałęczowianka 1L", 2.00m, DateTime.Parse("18-01-2022")),
        new Product("bułka pszenna 50g", 0.20m, DateTime.Parse("20-09-2021")),
        new Product("chleb krojony 600g", 2.45m, null)
    };

    int lengthLimit = 18;

    var allNamesAreCorrect = products
        .Where(x => x.Sold == false)
        .Select(x => x.Name)
        .All(x => false);

    if (allNamesAreCorrect)
    {
        Console.WriteLine("All names are correct.");
    }
    else
    {
        var invalidProducts = products
            .Where(x => x.Sold == false && x.Name.Length >= lengthLimit)
            .Count();

        Console.WriteLine($"Names too long: {invalidProducts}");
    }
}
```

Metoda OrderBy - zadanie

Bardzo przydatna umiejętność podczas pracy z kolekcjami to sortowanie. Oczywiście sortowanie można wykonać pisząc własny kod implementujący algorytm QuickSort albo MergeSort. Jednak dzięki LINQ nie musisz pisać go samodzielnie.

Wyobraźmy sobie, że w naszej bazie danych jest kilka produktów o tej samej nazwie. To często spotykany przykład bo możemy mieć na półce kilka butelek tego samego napoju. Sklep traci na przeterminowanych produktach sporo pieniędzy. Dlatego w wielu marketach znajdziemy półkę z towarami ze zbliżającą się datą przydatności. Za chwile wyszukamy produkty, które trzeba najpilniej sprzedać.

Będziemy szukać produktów, które niedługo będą przeterminowane. To one powinny zostać sprzedane jako pierwsze. Znajdziemy takie produkty za pomocą sortowania. Twoim zadaniem jest dodać metodę OrderBy aby po uruchomieniu programu na ekranie pojawił się taki komunikat:

Produkty należy sprzedawać w następującej kolejności:

ID: 1

NAZWA: Mleko UHT 1L

WAŻNY JESZCZE: 1 DNI

ID: 2

NAZWA: Mleko UHT 1L

WAŻNY JESZCZE: 19 DNI

ID: 3

NAZWA: Mleko UHT 1L

WAŻNY JESZCZE: 364 DNI

ID: 4

NAZWA: Mleko UHT 1L

WAŻNY JESZCZE: 364 DNI

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public DateTime? ValidTo { get; set; }
    public bool Sold { get; set; }

    public Product(int id, string name, decimal price, DateTime? validTo,
bool sold = false)
    {
        Id = id;
        Name = name;
        Price = price;
        ValidTo = validTo;
        Sold = sold;
    }
}

static void Main(string[] args)
{
    List<Product> products = new List<Product>()
    {
        new Product(3, "Mleko UHT 1L", 3.49m, DateTime.Now.AddYears(1)),
        new Product(2, "Mleko UHT 1L", 3.49m, DateTime.Now.AddDays(20)),
        new Product(4, "Mleko UHT 1L", 3.49m, DateTime.Now.AddYears(1)),
        new Product(1, "Mleko UHT 1L", 3.49m, DateTime.Now.AddDays(2))
    };

    var sortedProducts = products;
```

```
Console.WriteLine("Produkty należy sprzedawać w następującej kolejno-  
ści:");
```

```
foreach(var product in sortedProducts)  
{  
    var timeSpan = product.ValidTo - DateTime.Now;  
    Console.WriteLine($"ID: {product.Id}");  
    Console.WriteLine($"NAZWA: {product.Name}");  
    Console.WriteLine($"WAŻNY JESZCZE: {timeSpan.Value.Days} DNI");  
    Console.WriteLine();  
}  
}
```

Metoda OrderByDescending - zadanie

Ostatnie zadanie dotyczy metody **OrderByDescending**. Wyobraź sobie, że stworzysz system informatyczny na potrzeby konkursu dla programistów. Każdy uczestnik może zdobyć maksymalnie 100 punktów. Jednak możemy wyłonić tylko pięciu laureatów. To zadanie wydaje się skomplikowane, ale LINQ bardzo ułatwi nam rozwiązanie go.

Zadanie: Dodaj metodę **OrderByDescending** do poniższego kodu. Sam znajdź miejsce gdzie powinieneś ją dodać. Ustal też jaka będzie treść wyrażenia lambda. Naszym celem jest wyszukanie pięciu osób z najlepszym wynikiem egzaminu.

Jeśli dobrze wykonasz zadanie, to po uruchomieniu powinieneś zobaczyć następujący tekst:

Lista zwycięzców:

Maria Nowacka - 99 pkt.

Dariusz Kowalski - 90 pkt.

Anna Kowalska - 88 pkt.

Jan Nowak - 82 pkt.

Mariusz Nowak - 74 pkt.

```
public class ExamResult
{
    public string Name { get; set; }
    public int Points { get; set; }

    public ExamResult(string name, int points)
    {
        Name = name;
        Points = points;
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        var results = new List<ExamResult>();
        results.Add(new ExamResult("Jan Nowak", 82));
        results.Add(new ExamResult("Dariusz Kowalski", 90));
        results.Add(new ExamResult("Mariusz Nowak", 74));
        results.Add(new ExamResult("Jan Kwiatkowski", 51));
        results.Add(new ExamResult("Anna Kowalska", 88));
        results.Add(new ExamResult("Maria Nowacka", 99));
        results.Add(new ExamResult("Artur Kowal", 65));

        var winners = results
            .Take(5);

        Console.WriteLine("Lista zwycięzców:");

        foreach (var winner in winners)
        {
            Console.WriteLine($"{winner.Name} - {winner.Points} pkt.");
        }
    }
}
```

UWAGA: Pewnie zauważyłeś, że w przykładzie użyłem kolejnej metody LINQ. Ta metoda to **Take**. Działa ona podobnie do metody **First**. Z tą jednak różnicą, że pozwala na pobranie nie jednego ale kilku pierwszych elementów kolekcji. Ta metoda świetnie działa w połączeniu z **OrderBy** albo **OrderByDescending**. Taka kombinacja metod pozwala szybko wyszukać 3 najgorsze wyniki albo 5 największych liczb w kolekcji.

Podsumowanie

Mam nadzieję, że dużo się nauczyłeś na temat LINQ. Jeśli tutaj dotarłeś to znaczy, że najprawdopodobniej przeczytałeś część teoretyczną i wykonałeś wszystkie ćwiczenia z części praktycznej.

Nie omówiliśmy w tej książce wszystkich dostępnych metod LINQ. Ale moim zdaniem nie ma takiej potrzeby. Skupiliśmy się na najczęściej używanych metodach, a pozostałe znajdziesz w dokumentacji Microsoft.

Teraz kiedy już umiesz posługiwać się niektórymi metodami to bez problemu wykorzystasz w swojej pracy również te metody LINQ, których jeszcze nie znasz. Opanujesz też metody, które pojawią się w nowych wersjach platformy .NET. Przykładowo pod koniec roku 2021 zostanie wydana kolejna wersja o nazwie .NET 6. Wraz z nią pojawi się nowa metoda LINQ o nazwie **Chunk**. Będą też dostępne nowe wersje metod **ElementAt**, **Take** oraz **Zip**. Choć szansa, że będziesz z nich korzystał jest raczej niewielka, to z pewnością szybko je opanujesz kiedy zajdzie taka potrzeba.

Ta książka to powinien być dopiero początek Twojej nauki. Pamiętaj, że najlepiej opanujesz metody LINQ używając ich w swojej codziennej pracy. To co mogę Ci teraz doradzić to to, żebyś nie bał się korzystać z LINQ. Kiedy będziesz chciał napisać pętlę w swoim kodzie C# to zastanów się chwilę. Może zamiast pętli bardziej będzie opłacało się użyć jednej z metod LINQ?

Jeśli chcesz dowiedzieć się więcej na temat programowania w języku C# to koniecznie odwiedź moją stronę, albo wyślij mi maila. Chętnie Ci pomogę.

www.kursdotnet.pl

Dariusz Kacban
darek@kursdotnet.pl

