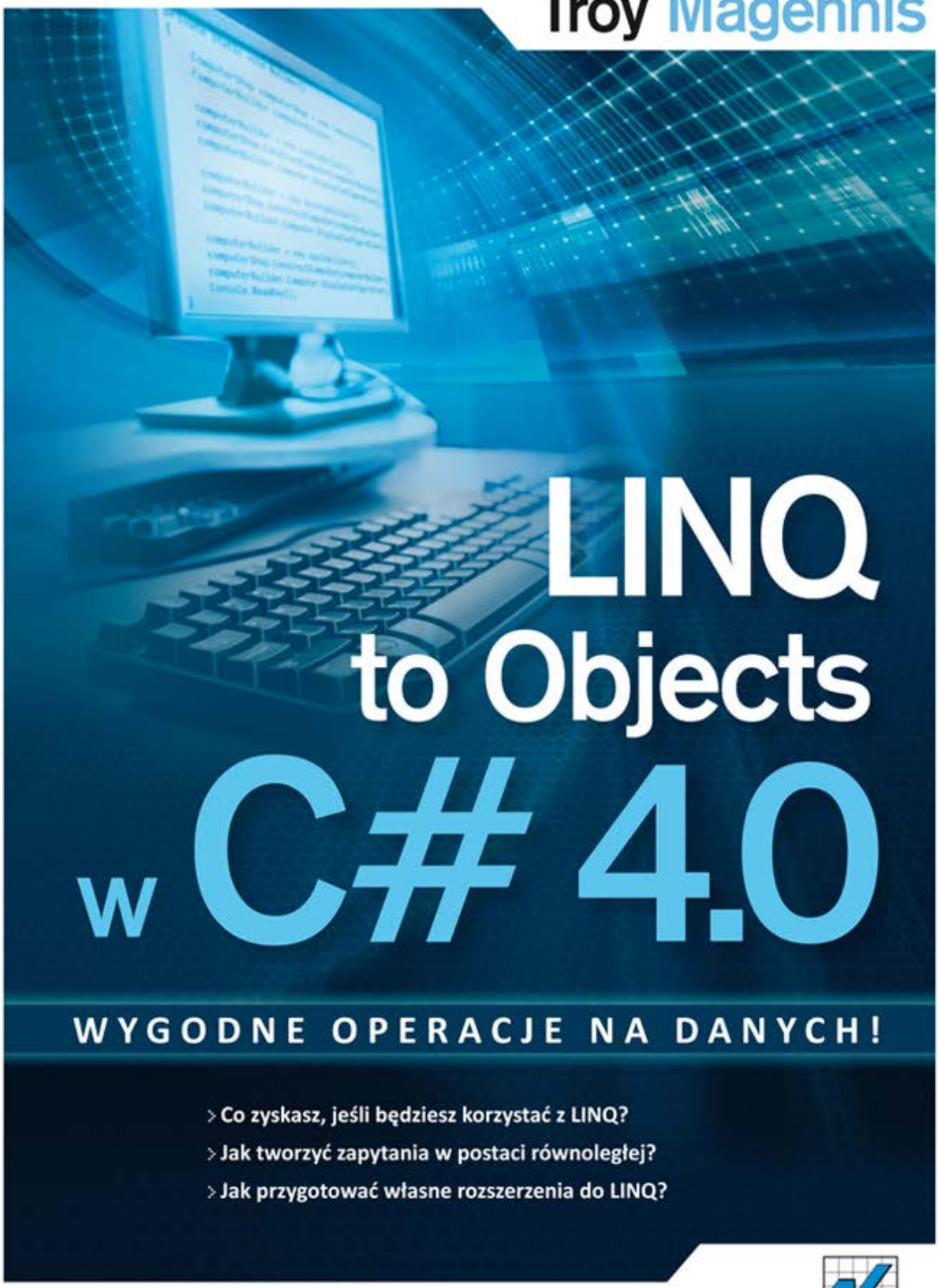
Troy Magennis



# LINQ to Objects w C# 4.0

WYGODNE OPERACJE NA DANYCH!

- Co zyskasz, jeśli będziesz korzystać z LINQ?
- Jak tworzyć zapytania w postaci równoległej?
- Jak przygotować własne rozszerzenia do LINQ?

Helion

Tytuł oryginału: LINQ to Objects Using C# 4.0: Using and Extending LINQ to Objects and Parallel LINQ (PLINQ)

Tłumaczenie: Łukasz Schmidt

ISBN: 978-83-246-3609-9

Authorized translation from the English language edition, entitled: LINQ TO OBJECTS USING C# 4.0: USING AND EXTENDING LINQ TO OBJECTS AND PARALLEL LINQ (PLINQ); ISBN 0321637003; by Troy Magennis, published by Pearson Education, Inc, publishing as Addison Wesley. 

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/linobj.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/linobj
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

# SPIS TREŚCI

ROZDZIAŁ 3

# PISANIE PODSTAWOWYCH ZAPYTAŃ

***Cele rozdziału:***

- Zaprezentowanie opcji składni LINQ.
- Wprowadzenie do pisania podstawowych zapytań.
- Pokazanie, w jaki sposób filtrować, dokonywać projekcji i sortować dane za pomocą zapytań LINQ.

Najważniejszym celem tego rozdziału jest przedstawienie podstaw pisania zapytań. W skład tych podstaw wchodzą informacje o opcjach pisania, filtrowaniu danych, sortowaniu i zwracaniu dokładnie takiego zestawu rezultatów, jakiego będziesz potrzebował. Po przeczytaniu tego rozdziału będziesz rozumiał, jak pisać najpopularniejsze elementy zapytań, a w kolejnym rozdziale rozbudujesz tę wiedzę, poznając bardziej zaawansowane funkcje zapytań służące do grupowania i złączania z innymi źródłami.

## Opcje stylów składni zapytań

Większość z wcześniejszych przykładów w tej książce używała składni wyrażeń zapytań, jednak istnieją dwa style pisania zapytań LINQ. Nie wszystkie operatory są dostępne w składni wyrażeń zapytań wbudowanej w kompilatorze C# i aby użyć pozostałych operatorów (lub wywoływać własne operatory), konieczne jest użycie składni zapytań w postaci metod rozszerzeń lub — jeśli to konieczne — połączenia obu składni. Będziesz musiał poznać oba style składni zapytań, aby pisać, odczytywać i rozumieć kod napisany za pomocą LINQ.

- **Format metody rozszerzenia** (zwany także składnią kropkową) — format metody rozszerzenia polega po prostu na zastosowaniu kolejno metod rozszerzeń, z których każda zwraca wynik `IEnumerable<T>`, pozwalający kolejnej metodzie „przepłynąć" na poprzednim wyniku itd. (tzw. *płynny interfejs*).

```
int[] nums = new int[] {0,4,2,6,3,8,3,1};
var result1 = nums.Where(n => n < 5).OrderBy (n => n);
// lub z podziałami linii dla większej przejrzystości
var result2 = nums
   .Where(n => n < 5)
   .OrderBy (n => n);
```

- **Format wyrażeń zapytań** (preferowany, w szczególności dla złączeń i grup) — mimo że nie wszystkie standardowe operatory są obsługiwane przez składnię wyrażeń zapytań, korzyści w postaci przejrzystego kodu dla obsługiwanych operatorów są bardzo duże. Składnia wyrażeń zapytań jest łatwiejsza od składni metod rozszerzeń, ponieważ upraszcza zapis poprzez usunięcie wyrażeń lambda i wprowadzenie znajomej, zbliżonej do SQL reprezentacji.

```
int[] nums = new int[] {0,4,2,6,3,8,3,1};

var result = (from n in nums
   where n < 5
   orderby n
   select n).Distinct();
```

- **Zapytania i format składni kropkowej** (połączenie dwóch formatów) — ten format łączy składnię wyrażeń zapytań, umieszczoną w nawiasach, za którą następują kolejne operatory zapisane za pomocą składni kropkowej. Tak długo, jak wyrażenie zapytania zwraca `IEnumerable<T>`, można za nim umieścić łańcuch metod rozszerzeń.

```
int[] nums = new int[] {0,4,2,6,3,8,3,1};

var result = (from n in nums
   where n < 5
   orderby n
   select n).Distinct();
```

---

**Której składni wyrażeń używać?** Wybór będzie zależny od osobistych preferencji, jednak celem jest używanie składni, która jest najłatwiejsza do odczytania i pomoże programistom, którzy mogą później pracować z kodem, zrozumieć Twoje intencje. Mając to na uwadze, staraj się nie mieszać niepotrzebnie różnych składni w jednym zapytaniu; wymieszanie stylów powoduje, że zapytanie trudniej odczytać, a czytający musi liczyć nawiasy, aby ustalić, do której części zapytania ma zastosowanie składnia metody rozszerzenia. Jeśli już łączysz style, umieszczaj je obok siebie, na przykład używaj wyrażenia zapytania na początku w nawiasach, następnie metod rozszerzeń na końcu, dla tych operatorów, dla których są potrzebne (tak jak pokazano we wszystkich przykładach w tej książce, w których łączenie składni było konieczne).

Ja preferuję (być może z tego powodu, że pracowałem z SQL) użycie składni wyrażeń zapytań wszędzie, gdzie jest to możliwe, oraz użycie składni metod rozszerzeń, kiedy konieczne jest zastosowanie operatora nieobsługiwanego przez wyrażenia zapytań (np. operatora `Distinct`), jednak zawsze umieszczam te operatory na końcu zapytania. Czasem używam wyłącznie składni metod rozszerzeń, ale nigdy w przypadku, kiedy zapytanie zawiera operatory `Join` lub `GroupBy`.

---

Każda ze składni zapytań ma swoje zalety i wady, które zostaną opisane szczegółowo w kolejnych podrozdziałach.

## Składnia wyrażeń zapytań

Składnia wyrażeń zapytań jest dostępna w wersji języka C# 3.0 i nowszych; sprawia, że zapytania są bardziej czytelne i spójne. Kompilator konwertuje wyrażenie zapytania na składnię metod rozszerzeń w czasie kompilacji, a więc wybór składni opiera się wyłącznie na kryterium czytelności.

Na rysunku 3.1 pokazano podstawową postać wyrażeń zapytań wbudowanych w C# 3.0.

```
IEnumerable<T>|T  zapytanie-wyrażenie-identyfikator =
    from identyfikator in wyrażenie
    let opcj. identyfikator = wyrażenie
    where opcj. wyrażenie-boolean
    join opcj. typ opcj. identyfikator in wyrażenie on
         wyrażenie equals wyrażenie into opcj. identyfikator
    orderby opcj. wyrażenie (a) sortujące ascending|descending opcj.
    group opcj. wyrażenie by wyrażenie into opcj. identyfikator
    select wyrażenie into opcj. identyfikator
```

**Rysunek 3.1.** Podstawowa forma składni wyrażeń zapytań. Specyfikacja języka C# 3.0 opisuje dokładnie, w jaki sposób ta forma przekładana jest na metody rozszerzeń w celu kompilacji

**Uwaga** Fakt, że kolejność słów kluczowych jest inna niż w SQL, nie jest szczęśliwy dla znawców SQL. Jednak ta różnica wynika z ważnej przyczyny, która ma ułatwić pracę programistom. Kolejność `From-Where-Select` pozwala środowisku programistycznemu (w tym przypadku Visual Studio) zapewnić pełne wsparcie Intellisense przy pisaniu zapytania. W chwili, kiedy wpiszesz wyrażenie `from`, zostaną wyświetlone właściwości danego elementu. To nie byłoby możliwe (i nie jest możliwe w przypadku narzędzi edycyjnych SQL Server), gdyby projektanci C# przyjęli bardziej znaną kolejność słów kluczowych `Select-From-Where`.

Większość składni wyrażeń zapytań nie wymaga wyjaśniania programistom, którzy mają doświadczenie w pracy z inną składnią zapytań, taką jak SQL. Mimo że kolejność jest inna od tradycyjnych języków zapytań, nazwa każdego słowa kluczowego daje wyraźną wskazówkę na temat jego działania, z wyjątkiem wyrażeń `let` i `into`, które zostaną opisane niżej.

### Let — tworzenie zmiennej lokalnej

Pisanie zapytań może często wymagać mniejszej ilości powielonego kodu, jeśli utworzymy zmienną lokalną, która posłuży do przechowania wyniku obliczenia pomocniczego lub podzapytania. Słowo kluczowe `let` pozwala przechować wynik wyrażenia (wartość lub podzapytanie) w zakresie zmiennej odpowiadającym pozostałej części pisanego zapytania. Po przypisaniu wartości po raz pierwszy, zmiennej nie można już przypisać innej wartości.

W poniższym kodzie do zmiennej lokalnej o nazwie average przypisywana jest wartość odpowiadająca wartości średniej całej sekwencji źródłowej, obliczanej jednokrotnie, ale wykorzystywanej w całej projekcji select każdego elementu:

```
var variance = from element in source
   let average = source.Average()
   select Math.Pow((element - average), 2);
```

Słowo kluczowe let implementowane jest wyłącznie przez kompilator, który tworzy typ anonimowy zawierający zarówno oryginalną zmienną zakresu (element w poprzednim przykładzie), jak i nową zmienną let. Powyższe zapytanie jest mapowane (przekładane przez kompilator) bezpośrednio na pokazane niżej zapytanie metody rozszerzenia:

```
var variance =
   source.Select (
      element =>
         new
         {
            element = element,
            average = source.Average ()
         }
      )
      .Select (temp0 =>
         Math.Pow (
            ((double)temp0.element - temp0.average)
            , 2));
```

Każda dodatkowa zmienna let, która zostanie wprowadzona, spowoduje, że bieżący typ anonimowy zostanie połączony z kolejnym typem anonimowym zawierającym siebie i dodatkową zmienną itd. Jednak cała ta „magia" staje się jasna, kiedy piszemy wyrażenie zapytania.

### Into — kontynuacja zapytania

Słowa kluczowe group, join i select wyrażeń zapytań umożliwiają przechwycenie otrzymanej sekwencji w zmiennej lokalnej i użycie jej w pozostałej części zapytania. Słowo kluczowe into pozwala kontynuować zapytanie poprzez użycie wyniku przechowanego w zmiennej lokalnej w każdym miejscu po jej definicji.

Najczęstszym sposobem wykorzystania into jest przechwycenie wyniku operacji grupowania, która razem z wbudowanymi funkcjami złączeń zostanie omówiona w rozdziale 4., „Grupowanie i łączenie danych". Poniższy przykład jest krótkim wprowadzeniem do tych funkcji. Grupuje wszystkie elementy o tej samej wartości i przechowuje wynik w zmiennej groups poprzez użycie słowa kluczowego into (w połączeniu ze słowem kluczowym group), zmienna groups może brać udział i jest dostępna w pozostałej części wyrażenia zapytania.

```
var groupings = from element in source
   group element by element into groups
   select new {
```

```
        Key = groups.Key,
        Count = groups.Count()
    };
```

## Porównanie opcji składni zapytań

Kod w listingu 3.1 używa składni metod rozszerzeń, a w listingu 3.2 składni wyrażeń zapytań, jednak funkcjonalnie są takie same: oba dają ten sam wynik, pokazany w wyjściu 3.1. Jasność kodu w składni wyrażenia zapytania bierze się z usunięcia zwrotów wyrażenia lambda i zastosowania stylu operatorów SQL. Oba style składni są funkcjonalnie identyczne i dla prostych zapytań (takich jak to) korzyść z uzyskania bardziej przejrzystego kodu jest minimalna.

**Listing 3.1.** Zapytanie wyszukuje wszystkie kontakty w województwie oznaczonym skrótem WM i sortuje je według nazwiska, a następnie imienia za pomocą składni metod rozszerzeń — zobacz wyjście 3.1

```
List<Contact> contacts = Contact.SampleData();

var q = contacts.Where(c => c.State == "WM")
   .OrderBy(c => c.LastName)
   .ThenBy(c => c.FirstName);

foreach (Contact c in q)
   Console.WriteLine("{0} {1}",
      c.FirstName, c.LastName);
```

**Listing 3.2.** To samo zapytanie, co w listingu 3.1, ale używające składni wyrażeń zapytań — zobacz wyjście 3.1

```
List<Contact> contacts = Contact.SampleData();

var q = from c in contacts
   where c.State == "WM"
   orderby c.LastName, c.FirstName
   select c;

foreach (Contact c in q)
   Console.WriteLine("{0} {1}",
      c.FirstName, c.LastName);
```

**Wyjście 3.1.**

```
Stanisław Kowal
Cyryl Latos
Alfred Wieczorek
```

Tym razem poprzez zastosowanie składni wyrażeń zapytań zamiast składni metod rozszerzeń uzyskujemy znaczne korzyści w postaci czytelności kodu; jest tak, kiedy zapytanie zawiera funkcje złączeń i (lub) grupowania. Mimo że nie cała funkcjonalność

grupowania i złączania jest bezpośrednio dostępna, kiedy używasz składni wyrażeń zapytań, to jednak większość pisanych przez Ciebie zapytań nie będzie wymagać tych dodatkowych funkcji. Listing 3.3 pokazuje dość nieczytelną składnię metod rozszerzeń dla `Join` (nieczytelną w tym znaczeniu, że po szybkim przeczytaniu kodu nie jest jasne, co oznacza każdy z argumentów metody `GropuBy`). Funkcjonalny odpowiednik korzystający ze składni wyrażeń zapytań dla tego samego zapytania został pokazany w listingu 3.4. Oba zapytania dają ten sam wynik, pokazany w wyjściu 3.2.

**Listing 3.3.** Złączenia stają się szczególnie złożone w składni metod rozszerzeń. To zapytanie zwraca pięć pierwszych szczegółowych zestawów informacji o połączeniach w kolejności od najnowszej — zobacz wyjście 3.2

```
List<Contact> contacts = Contact.SampleData();
List<CallLog> callLog = CallLog.SampleData();

var q = callLog.Join(contacts,
   call => call.Number,
   contact => contact.Phone,
   (call, contact) => new
   {
      contact.FirstName,
      contact.LastName,
      call.When,
      call.Duration
   })
.Take(5)
.OrderByDescending(call => call.When);

foreach (var call in q)
   Console.WriteLine("{0} - {1} {2} ({3}min)",
      call.When.ToString("ddMMM HH:m"),
      call.FirstName, call.LastName, call.Duration);
```

Jeśli to jeszcze nie stało się jasne, ja preferuję użycie składni wyrażeń zapytań za każdym razem, kiedy w zapytaniu wymagane jest wykonanie operacji `Join` lub `GroupBy`. Kiedy standardowy operator nie jest obsługiwany przez składnię wyrażeń zapytań (tak jak w przypadku np. metody `.Take`), umieszczam zapytanie w nawiasach i używam składni metod rozszerzeń od tego punktu, tak jak w listingu 3.4.

**Listing 3.4.** Składnia wyrażeń zapytań dla zapytania identycznego z tym pokazanym w listingu 3.3 — zobacz wyjście 3.2

```
List<Contact> contacts = Contact.SampleData();
List<CallLog> callLog = CallLog.SampleData();

var q = (from call in callLog
   join contact in contacts on
      call.Number equals contact.Phone
```

```
orderby call.When descending
select new
{
   contact.FirstName,
   contact.LastName,
   call.When,
   call.Duration
}).Take(5);

foreach (var call in q)
   Console.WriteLine("{0} - {1} {2} ({3}min)",
      call.When.ToString("ddMMM HH:m"),
      call.FirstName, call.LastName, call.Duration);
```

**Wyjście 3.2.**

```
07sie 11:15 - Stanisław Kowal (4min)
07sie 10:35 - Cezary Zbytek (2min)
07sie 10:5 - Maciej Karaś (1min)
07sie 09:23 - Adrian Hawrat (15min)
07sie 08:12 - Bartłomiej Gajewski (2min)
```

**Wskazówki dotyczące składni metod rozszerzeń**

- Na początku umieszczaj najbardziej ograniczającą metodę zapytania; dzięki temu zmniejszysz obciążenie kolejnych operatorów.
- Umieszczaj każdy operator w nowej linii (razem z kropką-łącznikiem). Dzięki temu będziesz mógł wykomentować poszczególne operatory podczas debugowania.
- Zachowaj spójność — w całej aplikacji używaj tego samego stylu.
- Aby ułatwić odczytywanie zapytań, nie bój się dzielenia zapytań na kilka części i stosowania wcięć do zaznaczenia ich hierarchii.

**Wskazówki dotyczące składni wyrażeń zapytań**

- Jeśli musisz wymieszać metody rozszerzeń z wyrażeniami rozszerzeń, umieść je na końcu.
- Umieść każdą część wyrażenia zapytania w oddzielnej linii. Dzięki temu będziesz mógł wykomentować poszczególne wyrażenia podczas debugowania.

## Jak filtrować wyniki (wyrażenie Where)

Jednym z najważniejszych zadań zapytania LINQ jest zawężenie wyników z większej kolekcji na podstawie określonych kryteriów. Można to osiągnąć za pomocą operatora `Where`, który testuje każdy element kolekcji źródłowej i zwraca tylko te elementy, które dają prawdziwy wynik sprawdzenia wobec kryteriów w wyrażeniu predykatu. **Predykat** (ang. *predicate*) to po prostu wyrażenie, które przyjmuje element tego samego typu, co elementy w kolekcji źródłowej, i zwraca prawdę lub fałsz. Predykat jest przekazywany do wyrażenia `Where` za pomocą wyrażenia lambda.

Metoda rozszerzenia dla operatora `Where` jest zaskakująco prosta; iteruje przez kolekcję za pomocą instrukcji `foreach`, testując każdy element i zwracając te, które przechodzą test. Oto kod zbliżony do kodu w bibliotece `System.Linq`:

```
public delegate TResult Func<T1, TResult>(T1 arg1);

public static IEnumerable<T> Where<T>(
   this IEnumerable<T> source,
   Func<T, bool> predicate) {

   foreach (T element in source) {
      if (predicate(element))
         yield return element;
   }
}
```

Operator `Where` w LINQ to Objects jest z pozoru rozwiązaniem bardzo podstawowym, ale jego implementacja jest prosta w wyniku zastosowania potężnej instrukcji `yield return`, która pojawiła się w platformie .NET 2.0, aby ułatwić budowanie iteratorów kolekcji. Każdy kod implementujący wbudowany wzorzec enumeracji (tak jak wszelkie kolekcje implementujące interfejs `IEnumerable`) samoistnie obsługuje obiekty proszące o kolejny element w kolekcji — jednocześnie przetwarzany jest kolejny element do zwrotu (obsługiwany np. przez słowo kluczowe `foreach`). Dowolna kolekcja implementująca wzorzec `IEnumerable<T>` (który także implementuje `IEnumerable`) będzie rozszerzana operatorem `Where`, który będzie zwracał pojedynczy element, kiedy zostanie o to poproszony, pod warunkiem że ten element będzie spełniał warunki wyrażenia predykatu (zwróci wartość `true`).

Wyrażenia filtrujące predykatów są przekazywane do metody rozszerzenia za pomocą wyrażenia lambda (krótki opis wyrażeń lambda znajdziesz w rozdziale 2., „Wprowadzenie do LINQ to Objects”), jednak jeśli zostanie użyta składnia wyrażenia zapytania, predykat filtrujący przybierze nawet bardziej elegancką formę. Oba style tworzenia wyrażeń predykatów zostaną opisane szczegółowo w kolejnych podrozdziałach.

### Filtr Where za pomocą wyrażenia lambda

W przypadku tworzenia predykatu dla operatora `Where` predykat przyjmuje element wejściowy tego samego typu, co elementy w kolekcji źródłowej, i zwraca prawdę lub

fałsz (wartość typu Boolean). Prosty predykat dla wyrażenia Where możesz zobaczyć w poniższym kodzie:

```
string[] animals = new string[] { "Koala", "Kangur",
   "Pająk", "Wombat", "Waran", "Wąż", "Rekin",
   "Płaszczka", "Meduza" };

   var q = from a in animals
      where a.StartsWith("W") && a.Length > 4
      select a;

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

W powyższym kodzie każda wartość ciągu znaków z tablicy nazw zwierząt jest przekazywana do metody rozszerzenia Where w zmiennej zakresu a. Każdy ciąg znaków w a jest sprawdzany na podstawie warunków funkcji predykatu i tylko te ciągi, które przechodzą test (zwracają prawdę), są zwracane w wynikach zapytania. W tym przykładzie tylko dwa ciągi znaków przechodzą test i są wyświetlane w oknie konsoli, są to:

```
Wombat
Waran
```

Kompilator C# konwertuje wyrażenie lambda na standardowe wywołanie metody anonimowej (poniższy kod jest funkcjonalnie taki sam):

```
var q = animals.Where(
   delegate(string a) {
   return a.StartsWith("W") && a.Length > 4; });
```

### Co to jest opóźnione wykonanie?

Wyrażenie Where rozpocznie testowanie predykatu dopiero wtedy, kiedy ktoś (czyli Ty poprzez wyrażenie foreach lub inny standardowy operator zapytania zawierający wyrażenie foreach) podejmie próbę iterowania przez wyniki; do tego czasu mechanizm iteratora będzie tylko pamiętał, w którym miejscu się znajdował, kiedy poproszono go po raz ostatni o element. Jest to tzw. **opóźnione wykonanie** (ang. *deferred execution*), sprawiające, że wykonanie zapytania jest przewidywalne i możesz kontrolować, kiedy i w jaki sposób zapytanie będzie wykonane. Jeśli potrzebujesz wyników natychmiast, możesz wywołać ToList(), ToArray() lub inny standardowy operator wywołujący natychmiastową aktualizację wyników w innej formie; w przeciwnym przypadku ewaluacja zapytania zostanie rozpoczęta dopiero, kiedy rozpoczniesz iterowanie przez jego wynik.

## Filtr Where za pomocą wyrażeń zapytań (sposób preferowany)

Składnia wyrażenia zapytania Where pomija wyraźną definicję zmiennej zakresu i operator wyrażenia lambda (=>), dzięki czemu składnia jest bardziej spójna i bardziej zbliżona do wyrażeń SQL, łatwo zrozumiałych dla wielu programistów. Z tych powodów jest to składnia preferowana. Przepisanie poprzedniego przykładu za pomocą składni wyrażeń zapytań pokazuje te różnice:

```
string[] animals = new string[] { "Koala", "Kangur",
   "Pająk", "Wombat", "Waran", "Wąż", "Rekin",
   "Płaszczka", "Meduza" };

var q = from a in animals
   where a.StartsWith("W") && a.Length > 4
   select a;

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

## Używanie metody zewnętrznej w celu ewaluacji

Mimo że możesz pisać zapytania zawierające bezpośrednio kod predykatu filtru, to nie musisz tego robić. Jeśli predykat jest długi i ma być wykorzystany w więcej niż jednym wyrażeniu zapytania, możesz rozważyć umieszczenie go we własnej metodzie (dobra praktyka dla wszelkiego powielanego kodu). Poprzedni przykład został przepisany za pomocą zewnętrznej funkcji predykatu, aby zademonstrować tę technikę:

```
string[] animals = new string[] { "Koala", "Kangur",
   "Pająk", "Wombat", "Waran", "Wąż", "Rekin",
   "Płaszczka", "Meduza" };

var q = from a in animals
   where MyPredicate(a)
   select a;

foreach (string s in q)
   Console.WriteLine(s);

public bool MyPredicate(string a)
{
   if (a.StartsWith("W") && a.Length > 4)
      return true;
   else
      return false;
}
```

Aby lepiej zademonstrować tę technikę w nieco bardziej skomplikowanym scenariuszu, kod w listingu 3.5 tworzy metodę predykatu, zawierającą logikę potrzebną do ustalenia, czy zwierzę „może być niebezpieczne". Poprzez enkapsulację tej logiki w pojedynczej metodzie nie musimy powielać tego samego kodu w kilku miejscach aplikacji.

**Listing 3.5.** Wyrażenie Where korzystające z metody zewnętrznej — zobacz wyjście 3.3

```
string[] animals = new string[] { "Koala", "Kangur", "Pająk", "Wombat", "Waran",
↪"Wąż", "Rekin", "Płaszczka", "Meduza" };

var q = from a in animals
```

```
   where IsAnimalDeadly(a)
   select a;
foreach (string s in q)
   Console.WriteLine("{0} może być niebezpieczny(a).", s);

public static bool IsAnimalDeadly(string s)
{
   string[] deadly = new string[] {"Pająk", "Waran", "Wąż", "Rekin", "Płaszczka",
   ↪"Meduza"};

   return deadly.Contains(s);
}
```

**Wyjście 3.3.**

```
Pająk może być niebezpieczny(a).
Waran może być niebezpieczny(a).
Wąż może być niebezpieczny(a).
Rekin może być niebezpieczny(a).
Płaszczka może być niebezpieczny(a).
Meduza może być niebezpieczny(a).
```

## Filtrowanie na podstawie pozycji indeksu

Standardowe operatory zapytań udostępniają wariant operatora `Where` uwidaczniający pozycję indeksu każdego przetwarzanego elementu kolekcji. Pozycja indeksu (rozpoczynająca się od zera) może być przekazana do wyrażenia lambda predykatu poprzez przypisanie nazwy zmiennej jako drugiego argumentu (po zmiennej zakresu elementu). Aby uwidocznić pozycję indeksu, należy użyć wyrażenia lambda, co można zrobić wyłącznie za pomocą składni metody rozszerzenia. Listing 3.6 pokazuje najprostszy przykład użycia — w tym przypadku polegający na zwróceniu pierwszego oraz wszystkich elementów o parzystych pozycjach indeksu w kolekcji źródłowej; wynik pokazano w wyjściu 3.4.

**Listing 3.6.** Pozycja indeksu może zostać użyta jako część predykatu wyrażenia Where, kiedy używamy wyrażeń lambda — zobacz wyjście 3.4

```
string[] animals = new string[] { "Koala", "Kangur", "Pająk", "Wombat",
"Waran", "Wąż", "Rekin", "Płaszczka", "Meduza" };

// pobranie pierwszego, a potem co drugiego zwierzęcia (indeksy nieparzyste)
var q = animals.Where((a, index) => index % 2 == 0);

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

**Wyjście 3.4.**

```
Koala
Pająk
Waran
Rekin
Meduza
```

## Jak zmienić typ zwracany (projekcja wybierania)

Kiedy piszesz zapytania SQL dla systemu baz danych, wskazywanie zestawu kolumn, z których mają zostać zwrócone wyniki, staje się Twoją drugą naturą. Celem jest ograniczenie zwracanych kolumn tylko do tych, które są niezbędne, aby poprawić wydajność i ograniczyć ruch w sieci (im mniej przesyłanych danych, tym lepiej). Efekt ten osiąga się poprzez wymienianie nazw kolumn po wyrażeniu `Select` w formacie pokazanym poniżej. W większości przypadków potrzebne kolumny są wskazywane za pomocą następującej składni SQL:

```
Select * from Contacts
Select ContactId, FirstName, LastName from Contacts
```

Pierwsze zapytanie zwróci każdą kolumnę (i wiersz) z tabeli `Contacts`, a drugie tylko trzy wskazane kolumny (dla każdego wiersza), dzięki czemu zaoszczędzona zostanie moc serwera i zasoby sieciowe. Konkluzją jest to, że składnia języka SQL pozwala na użycie odmiennego zestawu wierszy, który może nie odpowiadać istniejącej tabeli, widokowi lub schematowi jako strukturze zwracanych danych. Projekcje wybierania w wyrażeniach zapytań LINQ pozwalają nam osiągnąć to samo. Jeśli w zestawie wyników potrzebujemy tylko kilku wartości właściwości, zostaną zwrócone tylko te właściwości lub pola.

Projekcje wybierania w LINQ dają bardziej zróżnicowaną i efektywną kontrolę nad kształtem danych zwracanych z wyrażenia zapytania.

Sposoby zwracania wyników przez projekcję wybierania są następujące:

- Jako pojedyncza wartość wyniku lub element.
- W `IEnumerable<T>`, gdzie `T` jest tym samym typem, co elementy źródłowe.
- W `IEnumerable<T>`, gdzie `T` jest dowolnym istniejącym typem skonstruowanym w projekcji wybierania.
- W `IEnumerable<T>`, gdzie `T` jest typem anonimowym skonstruowanym w projekcji wybierania.
- W `IEnumberable<IGrouping<TKey, TElement>>`, który jest kolekcją pogrupowanych obiektów mających wspólny klucz.

Każdy styl projekcji ma swoje zastosowania i każdy zostanie opisany w kolejnych podrozdziałach.

**Jak wiele danych powinno być zawartych w projekcji wybierania?** Podobnie jak w przypadku wszystkich innych paradygmatów dostępu do danych, celem powinno być dążenie do zwracania tak małej liczby właściwości, jak to możliwe, kiedy definiujemy kształt wyniku zapytania. Dzięki temu zmniejsza się zużycie pamięci i dla wyniku łatwiej jest pisać kod, ponieważ istnieje w nim mniej właściwości, które należy uwzględnić.

## Zwracanie pojedynczej wartości wyniku lub elementu

Niektóre standardowe operatory zapytań zwracają jako wynik pojedynczą wartość lub pojedynczy element z kolekcji źródłowej. Operatory te zostały wymienione w tabeli 3.1. Każdy z nich kończy kaskadowanie wyników do innego zapytania i zamiast tego zwraca pojedynczą wartość wyniku lub element źródłowy.

**Tabela 3.1.** Przykładowy zestaw operatorów zwracających wynik będący wartością określonego typu (opisane w rozdziałach 5. i 6.)

| Zwracany typ | Operator |
|---|---|
| Numeryczny | `Aggregate`, `Average`, `Max`, `Min`, `Sum`, `Count`, `LongCount` |
| Boolean | `All`, `Any`, `Contains`, `SequenceEqual` |
| Typ<T> | `ElementAt`, `ElementAtOrDefault`, `First`, `FirstOrDefault`, `Last`, `LastOrDefault`, `Single`, `SingleOrDefault`, `DefaultIfEmpty` |

Jako przykład, poniższe proste zapytanie zwraca ostatni element w tablicy liczb całkowitych; wyświetla w oknie konsoli liczbę 2:

```
int[] nums = new int[] { 5, 3, 4, 2 };
int last = nums.Last();
ConsoleWriteLine(last);
```

## Zwracanie takiego samego typu jak źródło — IEnumerable<TŹródła>

Najbardziej podstawowy rodzaj projekcji zwraca przefiltrowany i posortowany podzestaw elementów oryginalnego źródła. Ta projekcja jest uzyskiwana poprzez wskazanie zmiennej zakresu jako argumentu po słowie kluczowym `select`. Kod w poniższym przykładzie zwraca `IEnumerable<Contact>`, którego typ — `Contact` jest ustalany na podstawie typu elementu w kolekcji źródłowej:

```
List<Contact> contacts = Contact.SampleData();
IEnumerable<Contact> q = from c in contacts
   select c;
```

Bardziej praktyczne zapytanie filtrowałoby wyniki i sortowało je w odpowiedni sposób. Zwracana jest ciągle kolekcja tego samego typu, ale liczba jej elementów i kolejność może być inna (niż kolekcji źródłowej).

```
List<Contact> contacts = Contact.SampleData();
IEnumerable<Contact> q = from c in contacts
   where c.State == "WA"
   orderby c.LastName,
      c.FirstName ascending
   select c;
```

## Zwracanie typu innego niż źródło — IEnumerable<TDowolny>

W projekcji za pomocą wyrażenia `select` może być zwrócony dowolny typ, nie tylko typ źródła. Typem docelowym może być dowolny dostępny typ, który mógłby być „ręcznie" skonstruowany za pomocą instrukcji `new` w zakresie pisanego kodu.

Jeśli tworzony typ posiada konstruktor z parametrami zawierający wszystkie parametry, jakich potrzebujesz, to wystarczy, że wywołasz ten konstruktor. Jeśli żaden z konstruktorów nie odpowiada parametrowi potrzebnemu dla tej projekcji, utwórz taki lub rozważ użycie składni C# 3.0 służącej do inicjalizacji typów (opisanej w rozdziale 2.). Korzyść wynikająca z użycia nowego rodzaju inicjalizatora polega na tym, że nie musisz definiować określonego konstruktora za każdym razem, kiedy potrzebna jest nowa sygnatura projekcji do obsłużenia zapytania w nowym kształcie. Kod w listingu 3.7 pokazuje, jak wykonać projekcję `IEnumerable<ContactName>` za pomocą obu konstruktorów.

**Listing 3.7.** Projekcja do kolekcji nowego typu — skonstruowanej za pomocą określonego konstruktora lub za pomocą składni inicjalizacji

```
List<Contact> contacts = Contact.SampleData();

// użycie konstruktora z parametrami
IEnumerable<ContactName> q1 =
   from c in contacts
   select new ContactName(
      c.LastName + ", " + c.FirstName,
      (DateTime.Now - c.DateOfBirth).Days / 365);

// użycie składni inicjalizowania typów
// uwaga: typ wymaga konstruktora bez parametrów
IEnumerable<ContactName> q2 =
from c in contacts
select new ContactName
{
   FullName = c.LastName + ", " + c.FirstName,
   YearsOfAge =
   (DateTime.Now - c.DateOfBirth).Days / 365
};

// definicja klasy ContactName
public class ContactName
{
   public string FullName { get; set; }
    public int YearsOfAge { get; set; }
```

```
    // konstruktor potrzebny dla przykładu inicjalizacji obiektów
    public ContactName() {
    }

    // konstruktor potrzebny dla przykładu projekcji typu
    public ContactName(string name, int age)
    {
       this.FullName = name;
       this.YearsOfAge = age;
    }
}
```

**Uwaga** Nie ulegaj pokusie zbyt częstego używania składni inicjalizatora. Wymaga ona, aby wszystkie właściwości inicjalizowane za jej pomocą mogły być odczytywane i zapisywane (posiadać instrukcje `get` i `set`). Jeśli właściwość ma być tylko do odczytu, nie zamieniaj jej we właściwość do odczytu i zapisu tylko po to, aby skorzystać z tej funkcji. W takim wypadku rozważ zmianę tych parametrów konstruktora na opcjonalne za pomocą składni C# 4.0 dla opcjonalnych parametrów konstruktora, tak jak opisano w rozdziale 8., „Nowe funkcje w C# 4.0".

## Zwracanie typu anonimowego — IEnumerable<TAnonimowy>

Typy anonimowe są nową funkcją języka wprowadzoną w C# 3.0; kompilator tworzy taki typ dynamicznie na podstawie wyrażenia inicjalizującego (wyrażenia po prawej stronie pierwszego znaku `=`). Jak już wyjaśniliśmy szczegółowo w rozdziale 2., nowemu typowi zostaje nadana przez kompilator nazwa niemożliwa do wywołania i bez słowa kluczowego `var` (zmienne lokalne z domyślnie określonym typem) skompilowanie zapytania nie byłoby możliwe. Poniższe zapytanie pokazuje projekcję do kolekcji `IEnumerable<T>`, gdzie `T` jest typem anonimowym:

```
List<Contact> contacts = Contact.SampleData();

var q = from c in contacts
   select new
   {
      FullName = c.LastName + ", " + c.FirstName,
      YearsOfAge =
         (DateTime.Now - c.DateOfBirth).Days / 365
};
```

Typ anonimowy utworzony w powyższym przykładzie składa się z dwóch właściwości: `FullName` i `YearsOfAge`.

Typy anonimowe uwalniają nas od potrzeby pisania i utrzymywania określonej definicji typu dla każdego odmiennego wyniku potrzebnego w kolekcji. Jedyną wadą jest to, że zakres typów tego rodzaju ograniczony jest do metody i nie mogą być one użyte

poza metodą, w której zostały zadeklarowane (chyba że zostaną przekazane jako typ `System.Object`, ale nie jest to sposób zalecany, ponieważ późniejszy dostęp do właściwości tego obiektu będzie wymagał użycia refleksji).

### Zwracanie zestawu pogrupowanych obiektów IEnumerable<IGrouping<TKlucz, TElement>>

W LINQ to Objects możliwe jest grupowanie wyników mających wspólne wartości źródłowe lub dowolną charakterystykę, którą można zrównać za pomocą wyrażenia używającego słowa kluczowego zapytania `group by` lub metody rozszerzenia `GroupBy`. Ten temat zostanie omówiony szczegółowo w rozdziale 4.

## Jak zwracać elementy, kiedy wynik jest sekwencją (SelectMany)

Standardowy operator zapytania `SelectMany` „spłaszcza" wszelkie elementy wyniku `IEnumerable<T>`, zwracając każdy element oddzielnie ze źródeł enumerowanych przed przejściem do kolejnego elementu w sekwencji wyniku. W przeciwieństwie do metody rozszerzenia `Select`, która zatrzyma się na pierwszym poziomie i zwróci sam element `IEnumerable<T>`.

Listing 3.8 pokazuje, w jaki sposób `SelectMany` różni się od `Select`; każda wersja kodu ma na celu pobranie pojedynczego słowa z zestawu przetwarzanych ciągów znaków. Aby pobrać słowa w opcji 1., wymagana jest podpętla `for`, ale `SelectMany` automatycznie wykonuje poditerację pierwotnej kolekcji wyników, tak jak pokazano w opcji 2. Opcja 3. pokazuje, że ten sam rezultat można osiągnąć za pomocą kilku wyrażeń `from` w wyrażeniu zapytania (które mapuje zapytanie tak, aby użyć operatora `SelectMany` w tle). Wynik w konsoli pokazano w wyjściu 3.5.

**Listing 3.8.** Select kontra SelectMany — SelectMany „zagłębia się" w wynik w postaci IEnumerable i zwraca jego elementy — zobacz wyjście 3.5

```
string[] sentence = new string[] { "Pewien szybki brązowy",
   "lis przeskakuje nad", "bardzo leniwym psem."};

Console.WriteLine("opcja 1:"); Console.WriteLine("------");

// opcja 2: Select zwraca trzy wartości string[]
// każda z nich zawiera trzy ciągi znaków.
IEnumerable<string[]> words1 =
   sentence.Select(w => w.Split(' '));

// aby pobrać każde ze słów, musimy użyć dwóch pętli foreach
foreach (string[] segment in words1)
   foreach (string word in segment)
      Console.WriteLine(word);
```

```
Console.WriteLine();
Console.WriteLine("opcja 2:"); Console.WriteLine("------");

// opcja 2: SelectMany zwraca dziewięć ciągów
// (iteruje przez wynik Select)
IEnumerable<string> words2 =
   sentence.SelectMany(segment => segment.Split(' '));

// za pomocą SelectMany mamy dostęp do każdego ciągu osobno
foreach (var word in words2)
   Console.WriteLine(word);

// opcja 3: identyczna jak opcja 2 powyżej napisana za pomocą
// składni wyrażeń zapytań (kilka instrukcji from)
IEnumerable<string> words3 =
   from segment in sentence
   from word in segment.Split(' ')
   select word;
```

**Wyjście 3.5.**

```
opcja 1:
------
Pewien
szybki
brązowy
lis
przeskakuje
nad
bardzo
leniwym
psem.

opcja 2:
------
Pewien
szybki
brązowy
lis
przeskakuje
nad
bardzo
leniwym
psem.
```

W jaki sposób działa metoda rozszerzenia `SelectMany`? Tworzy zagnieżdżoną pętlę `foreach` dla pierwotnego wyniku, zwracając każdy podelement za pomocą instrukcji `yield return`. Kod zbliżony do kodu `SelectMany` ma następującą postać:

```
static IEnumerable<S> SelectManyIterator<T, S>(
   this IEnumerable<T> source,
   Func<T, IEnumerable<S>> selector)
{
   foreach (T element in source)
   {
      foreach (S subElement in selector(element))
      {
         yield return podElement;
      }
   }
}
```

## W jaki sposób otrzymać wartości indeksu wyników

Select i SelectMany udostępniają przeciążenie, które umożliwia odczytanie wartości indeksów (rozpoczynających się od zera dla każdego zwracanego elementu w projekcji wybierania). Wartość indeksu jest uwidaczniana jako przeciążony parametr selektora wyrażenia lambda i jest dostępna wyłącznie za pomocą składni metody zapytania. Listing 3.9 pokazuje, w jaki sposób uzyskać dostęp do wartości indeksu w projekcji wybierania. Jak widać w wyjściu 3.6, ten przykład po prostu dodaje informację o pozycji na liście do każdego ciągu znaków będącego wynikiem wyboru.

**Listing 3.9.** Rozpoczynające się od zera wartości indeksu są uwidaczniane przez operatory Select i SelectMany — zobacz wyjście 3.6

```
List<CallLog> callLog = CallLog.SampleData();

var q = callLog.GroupBy(g => g.Number)
   .OrderByDescending(g => g.Count())
   .Select((g, index) => new
   {
      number = g.Key,
      rank = index + 1,
         count = g.Count()
   });

foreach (var c in q)
   Console.WriteLine(
      "Pozycja {0} - {1}, dzwonił(a) {2} razy.",
      c.rank, c.number, c.count);
```

**Wyjście 3.6.**

```
Pozycja 1 - 885 983 885, dzwonił(a) 6 razy.
Pozycja 2 - 546 607 546, dzwonił(a) 6 razy.
Pozycja 3 - 364 202 364, dzwonił(a) 4 razy.
Pozycja 4 - 603 303 603, dzwonił(a) 4 razy.
```

```
Pozycja 5 - 848 553 848, dzwonił(a) 4 razy.
Pozycja 6 - 165 737 165, dzwonił(a) 2 razy.
Pozycja 7 - 278 918 278, dzwonił(a) 2 razy.
```

## Jak usuwać powielone wyniki?

Standardowy operator zapytania `Distinct` powoduje, że zwracane są tylko niepowtarzalne wystąpienia w sekwencji. Ten operator przechowuje wewnętrznie informacje o elementach, które już zwrócił, i pomija drugie i kolejne wystąpienia danego elementu w trakcie zwracania. Operator zostanie omówiony szczegółowo w rozdziale 6., „Praca ze zbiorami danych", kiedy zajmiemy się jego użyciem w operacjach na zbiorach.

Operator `Distinct` nie jest obsługiwany przez składnię wyrażeń zapytań, a więc często umieszcza się go na końcu zapytania za pomocą składni metody rozszerzenia. Aby zademonstrować jego użycie, poniższy kod usunie powielone ciągi znaków. Wynik działania tego kodu w konsoli jest następujący:

```
Piotr
Paweł
Maria
Janina
```

```
string[] names = new string[] { "Piotr", "Paweł",
   "Maria", "Piotr", "Paweł", "Maria", "Janina" };

var q = (from s in names
   where s.Length > 3
   select s).Distinct();

foreach (var name in q)
   Console.WriteLine(name);
```

## Jak sortować wyniki

LINQ to Objects posiada rozbudowaną obsługę porządkowania i sortowania wyników. Niezależnie od tego, czy chcesz posortować w porządku rosnącym, malejącym na podstawie różnych wartości właściwości w dowolnej sekwencji, czy też napisać własny algorytm sortowania, funkcje sortowania w LINQ będą w stanie spełnić wszelkie wymagania.

### Podstawowa składnia sortowania

Kolekcja wyników otrzymana jako rezultat zapytania może być posortowana w dowolny pożądany sposób, w tym uwzględniający ustawienia regionalne oraz wielkość liter. W przypadku zapytań tworzonych za pomocą składni metod rozszerzeń proces sortowania obsługiwany jest przez standardowe operatory zapytań `OrderBy`, `OrderByDescending`, `ThenBy` i `ThenByDescending`. Operatory `OrderBy` i `ThenBy` sortują w porządku rosnącym

(np. od a do z), a operatory `OrderByDescending` i `ThenByDescending` w porządku malejącym (np. od z do a). Tylko pierwsza metoda rozszerzenia wykonująca sortowanie może używać operatorów `Orderby`, każde kolejne wyrażenie sortujące musi używać operatorów `ThenBy`, których może być zero lub więcej w zależności od tego, ile kontroli nad dalszym sortowaniem chcesz uzyskać, kiedy wiele elementów będzie mieć równoważne pozycje po wykonaniu poprzednich wyrażeń.

Poniższe przykłady pokazują sekwencję sortowania na początku według klucza `[w]`, a następnie w kolejności malejącej według klucza `[x]` i wreszcie w kolejności rosnącej według klucza `[y]`:

```
[source].OrderBy([w])
   .ThenByDescending([x])
   .ThenBy([y]);
```

Kiedy używamy składni wyrażeń zapytań, każdy klucz sortowania i opcjonalne słowo kluczowe wskazujące kierunek muszą być oddzielone przecinkami. Jeśli słowa `descending` lub `ascending` nie zostaną użyte, LINQ przyjmie kolejność rosnącą (`ascending`).

```
from [v] in [source]
orderBy [w], [x] descending, [y]
select [z];
```

Wynikiem sortowania kolekcji będzie `IOrderedEnumerable<T>`, który implementuje `IEnumerable<T>`, co umożliwi dalsze kaskadowanie operacji zapytań.

Metody rozszerzeń służące do sortowania zostały zaimplementowane za pomocą podstawowego, ale wydajnego algorytmu Quicksort (dokładniejsze wyjaśnienie jego działania — zobacz *http://en.wikipedia.org/wiki/Quicksort*). Implementacja w LINQ to Objects używa sortowania *niestabilnego*, co oznacza, że elementy odpowiadające tym samym wartościom klucza mogą nie zachować pozycji względem siebie, jaką miały w kolekcjach źródłowych (ten problem łatwo rozwiązać poprzez skierowanie wyniku do operatora `ThenBy` lub `ThenByDescending`). Algorytm jest dość szybki i znajduje zastosowanie w programowaniu równoległym, co zostało wykorzystane przez Microsoft w Parallel LINQ.

### Co to jest programowanie równoległe?

**Programowanie równoległe** (ang. *parallelization*) odnosi się do technik poprawiania wydajności aplikacji poprzez pełne wykorzystanie wielu procesorów i wielu rdzeni w procesorach wykonujących kod. Programowanie równoległe będzie omówione szczegółowo w rozdziale 9., „Parallel LINQ to Objects", w którym zostanie także pokazane, w jaki sposób zapytania LINQ mogą w pełni skorzystać z procesorów wielordzeniowych i wielu procesorów.

## Odwracanie kolejności sekwencji wyników (Reverse)

Inną metodą rozszerzenia związaną z sortowaniem, której działanie polega na odwróceniu kolejności całej sekwencji, jest `Reverse`. Jest ona wywoływana w postaci: `[źródło].Reverse();`. Ważnym zastrzeżeniem, o jakim należy pamiętać podczas używania operatora `Reverse`, jest to, że operator ten nie testuje równości elementów oraz nie wykonuje sortowania; po prostu zwraca elementy, zaczynając od ostatniego, a kończąc na pierwszym. Kolejność zwracania jest dokładnym odwróceniem kolejności, która zostałaby zwrócona z sekwencji wynikowej. Poniższy przykład pokazuje działanie operatora `Reverse` zwracającego litery `T O K` w oknie konsoli:

```
string[] letters = new string[] { "K", "O", "T" };
var q = letters.Reverse();
foreach (string s in q)
   Console.Write(" " + s);
```

## Sortowanie ciągów znaków bez uwzględniania wielkości liter oraz z użyciem ustawień regionalnych

Każdy standardowy operator zapytania, którego użycie wiąże się z sortowaniem, zawiera przeciążenie umożliwiające wskazanie określonej funkcji komparatora (przy zapisie za pomocą składni metod rozszerzeń). Biblioteka klas .NET zawiera przydatną klasę pomocniczą `StringComparer`, która posiada zestaw predefiniowanych, statycznych komparatorów gotowych do użycia. Komparatory te umożliwiają nam zmianę zachowania przy sortowaniu ciągów znaków, sterowanie uwzględnianiem wielkości liter oraz ustawieniami regionalnymi (ustawieniem języka dla bieżącego wątku). W tabeli 3.2 wymieniono statyczne instancje komparatorów, których można użyć w dowolnym operatorze `OrderBy` i `ThenBy` sortującym w porządku rosnącym bądź malejącym (zobacz także podrozdział „Własne komparatory równości dla operatorów zbiorów w LINQ" w rozdziale 6., który poświęcony jest wbudowanym komparatorom ciągów znaków i komparatorom własnym).

**Tabela 3.2.** Wbudowane funkcje StringComparer służące do sterowania z uwzględnieniem wielkości liter oraz sortowaniem z uwzględnieniem ustawień regionalnych

| Komparator | Opis |
|---|---|
| `CurrentCulture` | Wykonuje porównanie uwzględniające wielkość liter, używając reguł porównywania słów dla bieżących ustawień regionalnych. |
| `CurrentCultureIgnoreCase` | Wykonuje porównanie nieuwzględniające wielkości liter, używając reguł porównywania słów dla bieżących ustawień regionalnych. |
| `InvariantCulture` | Wykonuje porównanie uwzględniające wielkość liter, używając reguł porównywania słów dla niezmiennych ustawień regionalnych. |
| `InvariantCultureIgnoreCase` | Wykonuje porównanie nieuwzględniające wielkości liter, używając reguł porównywania słów dla niezmiennych ustawień regionalnych. |
| `Ordinal` | Wykonuje uwzględniające wielkość liter porównanie porządkowe. |
| `OrdinalIgnoreCase` | Wykonuje nieuwzględniające wielkości liter porównanie porządkowe. |

Listing 3.10 demonstruje składnię i efekt działania wbudowanych komparatorów dostępnych w .NET Framework. Wynik działania w konsoli pokazano w listingu 3.7, domyślny wynik uwzględniający wielkość liter można zmienić w nieuwzględniający ich wielkości.

**Listing 3.10.** Sortowanie uwzględniające wielkość liter oraz ustawienia regionalne/nieuwzględniające wielkości liter i ustawień regionalnych za pomocą funkcji StringComparer — zobacz wyjście 3.7

```
string[] words = new string[] {
   "jAnina", "JAnina", "janina", "Janina" };

var cs = words.OrderBy(w => w);
var ci = words.OrderBy(w => w,
   StringComparer.CurrentCultureIgnoreCase);

Console.WriteLine("Oryginalna kolejność:");
foreach (string s in words)
   Console.WriteLine(" " + s);

Console.WriteLine("Uwzględniające wielkości liter (domyślne):");
foreach (string s in cs)
   Console.WriteLine(" " + s);

Console.WriteLine("Nieuwzględniające wielkości liter:");
foreach (string s in ci)
   Console.WriteLine(" " + s);
```

**Wyjście 3.7.**

```
Oryginalna kolejność:
 jAnina
 JAnina
 janina
 Janina
Uwzględniające wielkości liter (domyślne):
 janina
 jAnina
 Janina
 JAnina
Nieuwzględniające wielkości liter:
 jAnina
 JAnina
 janina
 Janina
```

### Wskazywanie własnych funkcji komparatora

Aby obsłużyć wszelkiego rodzaju kolejności sortowania, istnieje możliwość łatwego wskazania własnych funkcji komparatora. Własna klasa komparatora będzie oparta na standardowym interfejsie .NET o nazwie `IComparer<T>` udostępniającym jedną metodą: `Compare`. Interfejs ten nie służy wyłącznie LINQ, jest podstawą wszystkich klas .NET wymagających możliwości sortowania (w tym na podstawie własnych kryteriów).

Funkcje komparatora zwracają wyniki w postaci liczby całkowitej, wskazującej na relację pomiędzy parą instancji typów. Jeśli dwa typy zostaną uznane za równe, funkcja zwraca zero. Jeśli pierwsza instancja jest mniejsza od drugiej, zwracana jest wartość ujemna; jeśli pierwsza instancja jest większa od drugiej, zwracana jest wartość dodatnia. To, w jaki sposób uzyskasz określone wyniki w postaci liczb całkowitych, zależy wyłącznie od Ciebie.

Aby zademonstrować działanie własnego `IComparer<T>`, kod w listingu 3.11 zawiera funkcję komparatora, która po prostu miesza (w sposób losowy) elementy ze źródła. Algorytm podejmuje losową decyzję o tym, czy dany element jest mniejszy czy większy od drugiego. Wyjście 3.8 pokazuje wynik w konsoli dla prostego źródła, którym jest zbiór ciągów znaków w tablicy; wynik będzie (potencjalnie) inny za każdym wykonaniem kodu.

**Listing 3.11.** Sortowanie za pomocą naszej własnej implementacji IComparer<T> w celu uzyskania losowych wyników — zobacz wyjście 3.8

```
public class RandomShuffleStringSort<T> : IComparer<T>
{
   internal Random random = new Random();

   public int Compare(T x, T y)
   {
      // liczba losowa: 0 lub 1
       int i = random.Next(2);

         if (i == 0)
            return -1;
         else
            return 1;
   }
}

string[] strings = new string[] { "1-jeden", "2-dwa",
   "3-trzy", "4-cztery", "5-pięć" };

var normal = strings.OrderBy(s => s);
var custom = strings.OrderBy(s => s,
   new RandomShuffleStringSort<string>());

Console.WriteLine("Zwykła kolejność sortowania:");
foreach (string s in normal) {
   Console.WriteLine(" " + s);
}
```

```
Console.WriteLine("Własna kolejność sortowania:");
   foreach (string s1 in custom) {
      Console.WriteLine(" " + s1);
}
```

**Wyjście 3.8.**

```
Zwykła kolejność sortowania:
 1-jeden
 2-dwa
 3-trzy
 4-cztery
 5-pięć
Własna kolejność sortowania:
 2-dwa
 5-pięć
 3-trzy
 4-cztery
 1-jeden
```

Częsty przypadek, który zawsze sprawiał mi kłopot, polega na tym, że proste sortowanie alfabetyczne nie zawsze prawidłowo przetwarza alfanumeryczne ciągi znaków. Przykładem jest sortowanie następujących ciągów: `File1`, `File10`, `File2`. Oczywiście, pożądaną kolejnością będzie `File1`, `File2`, `File10`, ale nie jest to kolejność alfabetyczna. Do osiągnięcia tego rezultatu konieczny będzie własny `IComparer`, który posortuje część alfabetyczną, a następnie numeryczną osobno. Jest to tzw. *sortowanie naturalne*.

Listing 3.12 i wyjście 3.9 pokazują własną klasę sortowania, która prawidłowo sortuje ciągi alfanumeryczne kończące się liczbami. W każdym przypadku, kiedy konieczne okaże się sortowanie tego rodzaju, nazwę klasy należy przekazać do dowolnej z metod rozszerzeń `OrderBy` lub `ThenBy` w następujący sposób:

```
string[] partNumbers = new string[] { "SCW10", "SCW1",
   "SCW2", "SCW11", "NUT10", "NUT1", "NUT2", "NUT11" };

var custom = partNumbers.OrderBy(s => s,
   new AlphaNumberSort());
```

Kod w listingu 3.12 na początku sprawdza, czy którykolwiek z wejściowych ciągów znaków ma wartość `null` lub jest pusty. Jeśli którykolwiek jest pusty, to kończy działanie i zwraca wynik z domyślnego komparatora (brak ciągu alfanumerycznego do sprawdzenia). Po ustaleniu, że istnieją dwa prawidłowe ciągi do porównania, dołączona część numeryczna każdego ciągu jest wydzielana w zmiennych `numericX` i `numericY`. Jeżeli żaden z ciągów nie ma części numerycznej, zwracany jest wynik z domyślnego komparatora (nie ma dołączonej części numerycznej dla jednego z ciągów, a więc wystarczające jest zwykłe porównanie). Jeśli oba ciągi zawierają część numeryczną, porównywana jest część nienumeryczna. Jeśli ciągi znaków są różne, zwracany jest wynik domyślnego komparatora (w takim przypadku część numeryczna nie ma znaczenia). Jeśli obie części

nienumeryczne są takie same, porównywane są wartości numeryczne `numericX` i `numericY` i ten wynik zostaje zwrócony. Ostatecznym wynikiem jest to, że elementy zostają posortowane alfabetycznie, a jeśli część literowa jest taka sama, o ostatecznej kolejności decyduje część numeryczna.

**Listing 3.12.** Sortowanie za pomocą własnego komparatora. Ten komparator prawidłowo sortuje ciągi znaków, które kończą się liczbą — zobacz wyjście 3.9

```
public class AlphaNumberSort : IComparer<string>
{
   public int Compare(string a, string b)
   {
      StringComparer sc =
         StringComparer.CurrentCultureIgnoreCase;

      // jeśli którakolwiek wartość wejściowa jest null lub pusta,
      // wykonaj proste porównanie ciągów
      if (string.IsNullOrEmpty(a) ||
         string.IsNullOrEmpty(b))
            return sc.Compare(a, b);

      // znajdywanie części numerycznych
      string numericX = FindTrailingNumber(a);
      string numericY = FindTrailingNumber(b);

      // jeśli w obu ciągach istnieje część numeryczna,
      // musimy badać dalej
      if (numericX != string.Empty &&
         numericY != string.Empty)
      {
         // na początku porównujemy przyrostek
         int stringPartCompareResult =
            sc.Compare(
               a.Remove(a.Length - numericX.Length),
               b.Remove(b.Length - numericY.Length));

         // jeśli ciągi stanowiące przyrostki są różne,
         // zwracamy wynik ich porównania
         if (stringPartCompareResult != 0)
            return stringPartCompareResult;

         // jeśli ciągi stanowiące przyrostki są takie same,
         // musimy sprawdzić także część numeryczną
         double nX = double.Parse(numericX);
         double nY = double.Parse(numericY);
         return nX.CompareTo(nY);
      }
      else
         return sc.Compare(a, b);
   }
```

```
private static string FindTrailingNumber(string s)
{
   string numeric = string.Empty;
   for (int i = s.Length - 1; i > -1; i--)
   {
      if (char.IsNumber(s[i]))
         numeric = s[i] + numeric;
      else
         break;
   }
   return numeric;
   }
}

string[] partNumbers = new string[] { "SCW10", "SCW1",
   "SCW2", "SCW11", "NUT10", "NUT1", "NUT2", "NUT11" };

var normal = partNumbers.OrderBy(s => s);
var custom = partNumbers.OrderBy(s => s,
   new AlphaNumberSort());

Console.WriteLine("Zwykły porządek sortowania:");
foreach (string s in normal)
   Console.WriteLine(" " + s);

Console.WriteLine("Własny porządek sortowania:");
foreach (string s in custom)
   Console.WriteLine(" " + s);
```

**Wyjście 3.9.**

```
Zwykły porządek sortowania:
 NUT1
 NUT10
 NUT11
 NUT2
 SCW1
 SCW10
 SCW11
 SCW2
Własny porządek sortowania:
 NUT1
 NUT2
 NUT10
 NUT11
 SCW1
 SCW2
 SCW10
 SCW11
```

**Uwaga** Aby osiągnąć ten sam wynik w większości systemów operacyjnych Windows (nie w Windows 2000, ale w ME, XP, 2003, Vista oraz Windows 7) i bez gwarancji, że nie zmieni się z czasem (zawiera następujące ostrzeżenie: „Zauważ, że działanie tej funkcji, a więc i zwracany przez nią wynik może zmieniać się w zależności od wersji. Nie powinna być używana w typowych aplikacjach sortujących"), Microsoft udostępnia API służący do sortowania plików w Explorerze (i być może w innych miejscach).

```
internal static class Shlwapi
{
   // http://msdn.microsoft.com/en-us/library/bb759947(VS.85).aspx
   [DllImport("shlwapi.dll", CharSet = CharSet.Unicode)]
   public static extern int StrCmpLogicalW(string a, string b);
}

public sealed class NaturalStringComparer : IComparer<string>
{
   public int Compare(string a, string b)
   {
      return Shlwapi.StrCmpLogicalW(a, b);
   }
}
```

## Podsumowanie

W tym rozdziale opisano podstawową funkcjonalność zapytań, polegającą na filtrowaniu, sortowaniu i projekcji wyników w dowolnej potrzebnej formie. Kiedy już zrozumiesz i opanujesz te podstawy zapytań, będziesz mógł sprawnie eksperymentować z bardziej zaawansowanymi funkcjami zapytań oferowanymi przez ponad 40 standardowych operatorów i w razie potrzeby pisać własne operatory.

# SKOROWIDZ

## F



## G



## H



## I



## J



## K

## P

## R



## S



## T

## U



## V



## W



## X



## Y



## Z

Współczesne systemy wymagają niezwykłej elastyczności, a przy tym muszą powstawać szybko. Klienci nigdy nie mają wystarczająco dużo czasu, żeby spokojnie poczekać na opracowywane rozwiązanie, a do tego wpadają na różne pomysły — w tym zmiany źródeł danych. LINQ to technologia, która pozwala na łatwe pobieranie danych z różnych źródeł dzięki wykonywaniu zapytań podobnych do zapytań SQL. Brzmi dobrze? Tak samo działa!

Ta książka porusza zagadnienia związane z jedną z gałęzi LINQ — LINQ to Objects. W trakcie lektury nauczysz się pisać podstawowe zapytania LINQ, filtrować wyniki, zwracać zestawy pogrupowanych obiektów oraz pobierać tylko unikalne wyniki. Ponadto dowiesz się, jak wykorzystać złączenia oraz używać z wbudowanych operatorów. Jako że jest to wyjątkowe kompendium wiedzy na temat LINQ to Objects, znajdziesz tu również szczegółowe informacje na temat tworzenia rozszerzeń do LINQ oraz opis zagadnień związanych z przetwarzaniem równoległym. Jeżeli zajmujesz się programowaniem w C# i chcesz skorzystać z nowoczesnych technologii, zainteresuj się tą książką!

- Historia LINQ
- Podstawowe zapytania — składnia
- Filtrowanie wyników — klauzula WHERE
- Rodzaje zwracanych wyników
- Sortowanie wyników
- Grupowanie i łączenie danych
- Rodzaje złączeń
- Dostępne operatory i ich wykorzystanie
- Operacje na zbiorach danych
- Tworzenie własnych operatorów
- Nowe funkcje w C# 4.0
- Typy dynamiczne w zapytaniach LINQ
- Równoległe wykonywanie zapytań
- Obsługa błędów

**WŁĄCZ NAJLEPSZE WZORCE LINQ TO OBJECTS DO TWOJEGO CODZIENNEGO PROGRAMOWANIA!**

helion.pl
księgarnia internetowa

Nr katalogowy: 7925

Księgarnia internetowa
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

Cena 49,00 zł

ISBN 978-83-246-3609-9

9 788324 636099