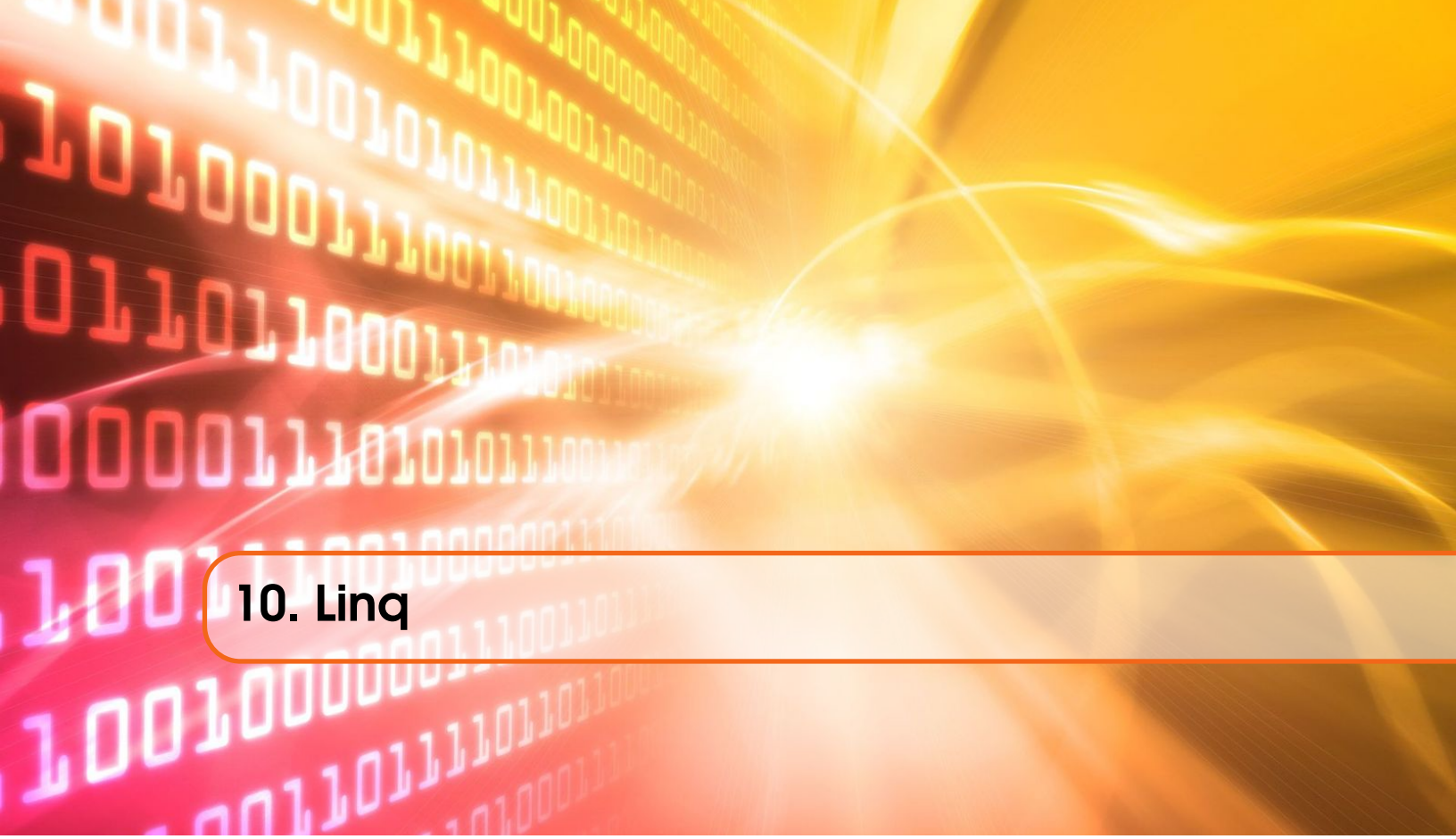
Programowanie funkcyjne

Materiały z wykładu i laboratorium

Łukasz Bartczuk



10. Linq

Linq jest to część platformy .NET, która jest odpowiedzialna za możliwość wydawania zapytań do obiektów. Pozwala on zapytania te zapisywać w dwojakiej formie. Albo jako zestaw wywołań funkcji:

```
int[] liczby = new [] { 0, 1, 2, 3, 4, 5, 6 };
int[] kwadratyParzystych = liczby.Where(l=>l%2==0)
                                 .Select(l=>l*l)
                                 .ToList();
```

albo w formie czytelnych zapytań w języku wzorowanym na składni SQL:

```
int[] liczby = new [] { 0, 1, 2, 3, 4, 5, 6 };
int[] kwadratyParzystych = (from l in liczby
                            where l%2==0
                            select l*l
                           ).ToList();
```

Bez względu na to, którą z tych form wybierzecie, skutek będzie dokładnie taki sam, gdyż zapytanie Linq (druga postać) zostanie przez kompilator przetworzona do wywołań funkcji (postać pierwsza).

Musicie pamiętać, że zapytania Linq dotyczące kolekcji działają na dowolnym jej typie o ile tylko kolekcja ta implementuje interfejs IEnumerable. Dlatego też zapytania te są wykonywane w sposób leniwy. Tzn. poniższy kod:

```
int[] liczby = new [] { 0, 1, 2, 3, 4, 5, 6 };
int[] kwadratyParzystych = from l in liczby
                           where l%2==0
                           select l*l;
```

spowoduje tylko przygotowanie ”przepisu” na działania jakie chcemy wykonać na naszej kolekcji obiektów. Dopiero wywołanie metody ToList (jak w poprzednim przykładzie) lub innej pobierającej choćby jeden element z tej kolekcji spowoduje uruchomienie zapytania.

10.1 Podstawy LINQ

W części podstawowej jest to zbiór metod rozszerzających interfejs IEnumberable<>. Zawarte są one w klasie Enumberable znajdującej sie w przestrzeni nazw System.Linq.

Zawiera ona wiele przydatnych metod, które możemy podzielić na kilka grup:

- **Agregacja**
  1. **`Aggregate`** – realizuje algorytm agregowania elementów kolekcji
  2. **`Average`** – Oblicza średnią wartość elementów w kolekcji
  3. **`Count`** – Określa liczbę elementów sekwencji
  4. **`Max`** – Zwraca wartość największą z kolekcji
  5. **`Max`** – Zwraca wartość najmniejszą z kolekcji
  6. **`Sum`** – Oblicza sumę elementów z kolekcji
- **Działania na elementach**
  1. **`DefaultIfEmpty`** – Zwraca określoną wartość jeżeli kolekcja jest pusta
  2. **`ElementAt`** – Zwraca element z określonego indeksu kolekcji. Rzuca wyjątkiem jeżeli indeks jest ujemny lub większy niż liczba elementów w kolekcji
  3. **`ElementAtOrDefault`** – Zwraca element z określonego indeksu kolekcji lub wartość domyślną dla danego typu, jeżeli indeks jest ujemy lub większy niż liczba elementów w kolekcji
  4. **`First`** – Zwraca pierwszy element sekwencji
  5. **`FirstOrDefault`** – Zwraca pierwszy element sekwencji. Jeżeli sekwencja jest pusta zwraca domyślną wartość dla danego typu
  6. **`Last`** – Zwraca ostatni element sekwencji
  7. **`LastOrDefault`** – Zwraca ostatni element sekwencji. Jeżeli sekwencja jest pusta zwraca domyślną wartość dla danego typu
  8. **`Single`** – Zwraca pierwszy element z kolekcji, ale pod warunkiem, że ta zawiera tylko jeden element.
  9. **`Single`** – Zwraca pierwszy element z kolekcji, ale pod warunkiem, że ta zawiera tylko jeden element. Jeżeli kolekcja zawiera więcej niż jeden element, lub jest pusta zwraca wartość domyślną dla danego typu
- **Filtrowanie**
  1. **`OfType`** – Filtruje elementy przepuszczając te z określonym typem
  2. **`Where`** – realizuje algorytm filtrowania kolekcji
- **Generowanie**
  1. **`Append`** – Dodaje nowy element na koniec sekwencji elementów kolekcji
  2. **`Concat`** – Dodaje na koniec sekwencji elementów pierwszej kolekcji, elementy drugiej

3. **`Empty`** – Zwraca pustą kolekcję policzalną
4. **`Prepend`** – Dodaje nowy element na koniec sekwencji elementów kolekcji
5. **`Range`** – Zwraca sekwencję n liczb całkowitych począwszy od określonej wartości
6. **`Repeat`** – Tworzy sekwencję elementów z n-krotnego powtórzenia określonej wartości
7. **`Zip`** – Łączy dwie sekwencje elementów w jedną sekwencję par

- **Grupowanie**
  1. **`GroupBy`** – Grupuje elementy zgodnie z określonym kluczem

- **Konwersje**
  1. **`Cast`** – Rzutuje elementy kolekcji IEnumerable (niegenerycznej) na określony typ
  2. **`ToArray`** – Tworzy tablicę z elementów sekwencji wejściowej
  3. **`ToDictionary`** – Tworzy słownik z elementów sekwencji wejściowej
  4. **`ToHashSet`** – Tworzy zbiór z elementów kolekcji wejściowej
  5. **`ToList`** – Tworzy listę z elementów kolekcji wejściowej
  6. **`ToLookup`** – Tworzy kolekcję umożliwiającą efektywne wyszukiwanie elementów na podstawie klucza. Działanie bardzo zbliżone do grupowania. Należy jednak pamiętać, że grupowanie realizowane jest w sposób leniwy, a ta metoda działa w sposób zachłanny
- **Operacje na zbiorach**
  1. **`Distinct`** – Zwraca różne elementy z kolekcji
  2. **`Exept`** – Określa różnicę zbiorów dwóch sekwencji
  3. **`Intersect`** – Realizuję operacje przecięcia dwóch sekwencji
  4. **`Union`** – Oblicza sumę zbiorów z dwóch sekwencji wejściowych
- **Podział**
  1. **`Skip`** – Pomija pierwsze n elementów z sekwencji
  2. **`SkipLast`** – Pomija ostatnie n elementów z sekwencji
  3. **`SkipWhile`** – Pomija pierwsze elementy z kolekcji, dopóki te spełniają określony warunek
  4. **`Take`** – Zwraca sekwencję n pierwszych elementów z sekwencji wejściowej
  5. **`TakeLast`** – Zwraca sekwencję n ostatnich elementów z sekwencji wejściowej
  6. **`TakeWhile`** – Zwraca sekwencję pierwszych elementów z sekwencji wejściowej, dopóki te spełniają określony warunek
- **Projekcja**
  1. **`Select`** – realizuje algorytm mapowania kolekcji
  2. **`SelectMany`** – Rzutuje każdy element kolekcji na nową sekwencję elementów, spłaszcza wynikowe sekwencje w jedną i rzutuje element każdy element tej sekwencji na określoną wartość
- **Sprawdzanie warunków**
  1. **`Contains`** – Określa czy sekwencja zawiera określony element
  2. **`All`** – sprawdza czy podany jako parametr predykat jest spełniony dla wszystkich elementów kolekcji
  3. **`Any`** – sprawdza czy podany jako parametr predykat jest spełniony dla któregokolwiek z elementów kolekcji
  4. **`SequenceEqual`** – Określa równość dwóch sekwencji - czy sekwencje zawierają te same wartości na tych samych pozycjach
- **Porządkowanie**
  1. **`OrderBy`** – Sortuje elementy kolekcji w kolejności rosnącej
  2. **`OrderByDescending`** – Sortuje elementy kolekcji w kolejności malejącej
  3. **`Reverse`** – Odwraca kolejność elementów w sekwencji

4. **`ThenBy`** – Określa kolejny porządek sortowania elementów w kolekcji w kolejności rosnącej
5. **`ThenByDescending`** – Określa kolejny porządek sortowania elementów w kolekcji w kolejności malejącej

- **Złączenia**
  1. **`GroupJoin`** – Łączy odpowiadające sobie elementy dwóch kolekcji i grupuje wyniki
  2. **`Join`** – Łączy elementy dwóch sekwencji bazując na dopasowaniu kluczy

Jak widzimy, funkcje te odpowiadają funkcjom F# z modułów List, czy Array, chodź nie da się ukryć, że zestaw funkcji F# jest bogatszy.

W celu przedstawienia działania niektórych z tych funkcji posłużę się następującymi klasami:

```
class Osoba
{
  public int Id { get; init; }
  public string Imie { get; init; }
  public string Nazwisko { get; init; }
}

class Pojazd
{
  public int    IdOsoby { get; init; }
  public string Pesel { get; init; }
}
```

Wykorzystując je stwórzmy dwie kolekcje, które wykorzystamy w zapytaniach Linq:

```
var osoby = new []
{
  new Osoba(1, "Ala", "Kot"),
  new Osoba(2, "Cezary", "Adamski"),
  new Osoba(3, "Franek", "Relke"),
  new Osoba(4, "Tomasz", "Nowak"),
  new Osoba(5, "Aleksander", "Nowak")
};

var pojazdy = new []
{
  new Pojazd {IDOsoby = 1, NrRej="SC12345"},
  new Pojazd {IDOsoby = 2, NrRej="WE12345"},
  new Pojazd {IDOsoby = 3, NrRej="KW12345"}
};
```

Operacja projekcji - select

Metoda Select realizuje dobrze znaną wam operację mapowania. Jej działanie (oczywiście w uproszczeniu można przedstawić jako):

```
public static IEnumerable<TResult> Select<TSource, TResult>
   ( this IEnumerable<TSource> source,
```

```
        Func<TSource, TResult> selector )
{
  foreach(var s in source)
    yield return selector(s);
}
```

Jak łatwo zauważyć, nie różni się ona zbytnio od funkcji, które tworzyliśmy na poprzednich zajęciach. Podobnie będzie przy pozostałych funkcjach Linq działających na obiektach. W końcu wykorzystują one iteratory.

Najprostsze możliwe wykorzystanie tej metody wymaga przekazania do niej funkcji tożsamościowej[1]:

```
var iterator = osoby.Select(x=>x);
```

Spowoduje to utworzenie iteratora, który przejdzie po wszystkich elementach kolekcji. W formie zapytania Linq powyższą instrukcję możemy zapisać jako:

```
var iterator = from osoba in osoby select osoba;
```

Oczywiście powyższe samodzielne wywołania funkcji lub klauzuli select (na kolekcji obiektów znajdującej się w pamięci) nie mają większego praktycznego zastosowania. Musimy jednak pamiętać, że o ile w tym przypadku metodę Select możemy pominąć, jeżeli korzystamy z funkcji, to w przypadku zapytania klauzula select musi wystąpić prawie zawsze.

Bardziej praktycznym zastosowaniem klauzuli select jest modyfikowanie obiektów przed ich zwróceniem (czyli wykonanie mapowania). Przykładowo, jeżeli będziemy chcieli utworzyć kolekcję obiektów zawierających tylko imię i nazwisko osoby, zamiast wszystkich informacji jakie są w niej zawarte możemy to zrobić następująco:

```
var tylkoImionaINazwiska = osoba.Select(
     osoba=>new {
                  osoba.Imie,
                  osoba.Nazwisko
                }
    );
```

lub w formie zapytania:

```
var tylkoImionaINazwiska = from osoba in osoby
                           select new {
                                        osoba.Imie,
                                        osoba.Nazwisko
                                      };
```

Zwróćmy uwagę, że w powyższym zapytaniu po słowie kluczowym new nie pojawia się nazwa klasy, której obiekt chcemy stworzyć. Oznacza to, że utworzony zostanie obiekt klasy anonimowej. Nazwa takiej klasy jest generowana przez kompilator i programista nie ma do niej dostępu. Wszystkie właściwości tej klasy są definiowane, jako tylko do odczytu. Jest to przydatne, jeżeli chcemy ograniczyć liczbę dostępnych pól,

[1]Musicie pamiętać, że funkcje te wykonywane są w sposób leniwy

bez konieczności deklarowania nowego typu. Trzeba pamiętać jednak, że obiekty te musimy skonsumować w tym samym bloku, w którym są zdefiniowane.

Operacja projekcji - SelectMany

Opisana wyżej operacja Select sprawdza się, jeżeli zastosowana w niej funkcja selektor zwraca w rezultacie pojedynczą wartość. Co jednak będzie, jeżeli funkcja ta zwróci nam inną kolekcję, np. dla każdej osoby w kolekcji osoba będzie zwracała wszystkie należące do niej pojazdy:

```
var ps = from osoba in osoby
         select pojazdy
                .Where(pojazd => pojazd.IdOsoby == osoba.Id);
```

W rezultacie uzyskamy oczywiście policzalną kolekcję, której elementami są inne policzalne kolekcje. Czasem jednak będziemy chcieli spłaszczyć tę strukturę tak aby wszystkie zwrócone kolekcje zostały złączone w jedną. Możemy to zrealizować za pomocą operacji SelectMany, którą w formie zapytania zapisuje się w formie:

```
var ps = from osoba in osoby
         from pojazd in
           pojazdy.Where(pojazd => pojazd.IdOsoby == osoba.Id)
         select pojazd;
```

Powyższe zapytanie możemy zapisać również z wykorzystaniem funkcji:

```
var ps = osoby.SelectMany(
          osoba => pojazdy
                   .Where(pojazd => pojazd.IdOsoby == osoba.Id));
```

Jeżeli przyjrzymy się dokładniej sposobowi działania tej funkcji, to bardzo szybko zorientujemy się, że odpowiada ona znanej nam już z programowania funkcyjnego operacji **bind**.

Operacja selekcji - Where

Metoda where służy do zdefiniowania operacji filtrowania. Podobnie jak w przypadku metody select jej definicja jest dosyć prosta i może być przedstawiona w następującej postaci:

```
public static IEnumerable<TSelect> Where<TSource>
    ( this IEnumerable<TSource> source,
      Func<TSource, bool> predicate )
  {
    foreach(var s in source)
      if (predicate(s))
          yield return selector(s);
  }
```

Przykładowo, jeżeli chcemy wybrać wszystkie pojazdy, których numer rejestracyjny rozpoczyna się od liter ”SC”, możemy zrealizować to następująco, w formie zapytania i wywołań funkcji:

```
var pojazdyZCzwy = from pojazd in pojazdy
                   where pojazd.NrRej.StartsWith("SC")
                   select pojazd;
```

```
var pojazdyZCzwy = pojazdy
                   .Where(pojazd=>pojazd.NrRej.StartsWith("SC"));
```

Zwróćmy uwagę, że w przypadku stosowania metody Where nie ma konieczności wywoływania po niej metody Select.

Operacja grupowania - Group By

Klauzula group by, podobnie jak w bazach danych realizuje operację grupowania. Załóżmy, że chcemy podzielić pojazdy na grupy względem identyfikatora osoby do której należą. W formie zapytania zapiszemy to w następującej formie:

```
var pojazdyOsob = from pojazd in pojazdy
                  group pojazd by pojazd.IdOsoby;
```

W powyższym przykładzie istotne jest, że w przypadku stosowania takiego prostego grupowania po klauzuli group by nie może wystąpić klauzula select. Powyższe zapytanie zapisane w formie wywołań metod będzie wyglądało następująco:

```
var pojazdyOsob = pojazdy.GroupBy(pojazd=>pojazd.IdOsoby);
```

W obu przypadkach do zmiennej pojazdyOsob zostanie przypisana policzalna kolekcja obiektów implementujących interfejs IGrouping o następującej definicji:

```
public interface IGrouping<out TKey, out TElement>
  : IEnumerable<TElement>, IEnumerable
{
  TKey Key { get; }
}
```

Jak widzimy jest to rozszerzenie interfejsu IEnumerable<> dodające do niego właściwość zawracającą klucz grupowania.

Każdy element kolekcji zwróconej przez metodę GroupBy jest wobec tego kolekcją, która zawiera tylko te obiekty, które należą do tej samej grupy (mają taki sam klucz grupowania). Oczywiście w tym przypadku typ tych obiektów jest dokładnie taki sam jak w kolekcji wejściowej.

Jeżeli chcielibyśmy teraz np. wyświetlić te wszystkie informacje na ekranie, to możemy to zrobić np. w następujący sposób:

```
foreach(var grupa in pojazdyOsob)
{
  Console.WriteLine($"Osoba {grupa.Key} posiada pojazdy:");
  foreach(var pojazd in grupa)
    Console.WriteLine(pojazd.NrRej);
}
```

Czasami na każdej grupie będziemy chcieli wykonać jakieś działania. Przykładowo, zamiast zwrócić kolekcję obiektów z każdej grupy, będziemy chcieli obliczyć jej liczebność. Aby to uczynić, to w przypadku zapytania, każdą grupę powinniśmy przypisać do zmiennej, co pozwoli nam zastosować dalej klauzulę select:

```
var pojazdyOsob = from pojazd in pojazdy
                  group pojazd by pojazd.IdOsoby into grupa
                  select new {
                               IDOsoby = grupa.Key,
                               LiczbaPojazdow = grupa.Count()
                             };
```

W przypadku stosowania funkcji GroupBy możemy wykorzystać jej wersję, która oprócz funkcji określającej klucz grupowania przyjmuje funkcję, która będzie mapowała każdą grupę:

```
var pojazdyOsob = pojazdy
                  .GroupBy(
                     g=>g.IdOsoby,
                     (idOsoby, pojazdy) =>
                        new {
                              IDOsoby = idOsoby,
                              LiczbaPojazdow = pojazdy.Count()
                            }
                   );
```

Zwróćmy uwagę, że w tym przypadku funkcja GroupBy ukrywa przed programistą interfejs IGrouping<,> nie przekazując go bezpośrednio do funkcji, tylko rozbijając go na osobne części: klucz grupowania oraz kolekcję obiektów należących do danej grupy.

Operacja złączenia - Join

Operacja Join pozwala nam na zrealizowanie złączenia dwóch kolekcji. Przykładowo, jeżeli chcielibyśmy wyświetlić imię i nazwisko osoby oraz numer rejestracyjny pojazdu, który do niej należy, zrobimy to w następujący sposób:

```
var osobyIPojazdy =
  from osoba in osoby
  join pojazd in pojazdy on osoba.Id equals pojazd.IdOsoby
  select new { osoba.Imie, osoba.Nazwisko, pojazd.NrRej };
```

W zapytaniu najpierw musimy zapisać klauzulę from, która spowoduje przejście po wszystkich elementach kolekcji zewnętrznej. Następnie dla każdego z nich z kolekcji wewnętrznej będą wybrane te elementy z drugiej kolekcji, które będą spełniały warunek złączenia. Zapisuje się to w klauzurze join on, w której podajemy drugą kolekcję oraz warunek złączenia. Na samym końcu w klauzurze select określamy sposób w jaki zwrócimy wyniki.

Powyższe zapytanie możemy przedstawić również w formie funkcji Join, która jako parametry przyjmuje obie kolekcje, funkcje, które pozwolą określić klucze złączenia osobno dla każdej z nich oraz funkcję mapującą wyniki:

```
var osobyIPojazdy =
      osoby.Join(pojazdy,
                 o => o.Id,
                 p => p.IdOsoby,
                 (o, p) => new { o.Imie, o.Nazwisko, p.NrRej }
                );
```

Złączenia grupujące - GroupJoin

Złączenie grupujące łączy w sobie dwie omawiane wcześniej operacje złączenia i grupowania. Omawiając operację grupowania przedstawiłem przykład, który dzielił nam kolekcję pojazdów na grupy względem identyfikatora jego posiadacza. Jeżeli zamiast identyfikatora chcielibyśmy uzyskać imię i nazwisko tej osoby powinniśmy najpierw wykonać operację złączenia, a następnie grupować te elementy względem określonego klucza. Metoda GroupJoin pozwala to zapisać w formie jednego wywołania funkcji, przy czym grupowanie zawsze odbywa się wg. klucza złączenia.

Jako zapytanie zapisujemy to w formie:

```
var osobyIPojazdy =
  from osoba in osoby
  join pojazd in pojazdy on osoba.Id equals pojazd.IdOsoby
  into grupa
  select new { osoba.Imie,
               osoba.Nazwisko,
               pojazdy = grupa.ToList()
             };
```

W przypadku korzystania z metod, jedyną różnicą jaka występuje pomiędzy Join i GroupJoin jest nazwa metody. Parametry obu są dokładnie takie same:

```
var osobyIPojazdy =
  osoby.GroupJoin(pojazdy,
                  o => o.Id,
                  p => p.IdOsoby,
                  (osoba, pojazdy) =>
                     new { osoba.Imie,
                           osoba.Nazwisko,
                           Pojazdy = pojazdy.ToList()
                         }
                 );
```

Złączenia zewnętrzne

Operacja join realizuje tzw. złączenie wewnętrzne tzn. dla każdego obiektu znajdującego się w zewnętrznej kolekcji stara się odnaleźć odpowiadający mu obiekt w kolekcji wewnętrznej. Jeżeli takiego obiektu nie uda się znaleźć, obiekt zewnętrzny nie jest umieszczany w kolekcji wynikowej. Czasami jednak będziemy chcieli, aby w rezultacie uzyskać wszystkie obiekty z kolekcji zewnętrznej. W takim przypadku dla wszystkich właściwości pochodzących z obiektów wewnętrznych podstawiona będzie wartość domyślna lub null. Taki rodzaj złączenia będziemy nazywać złączeniem zewnętrznym lewostronnym. Możemy je zrealizować za pomocą następującego zapytania:

```
var osobyIPojazdy =
  from osoba in osoby
  join pojazd in pojazdy on osoba.Id equals pojazd.IdOsoby
  into grupy
  from g in grupy.DefaultIfEmpty()
  select new { osoba.Imie,
               osoba.Nazwisko,
               NrRej = g?.NrRej ?? "Brak pojazdu"
             };
```

Operacji tej nie można zapisać za pomocą jednej funkcji i wymaga ona zastosowania zarówno metody GroupJoin oraz SelectMany:

```
var osobyIPojazdy =
  osoby.GroupJoin(pojazdy,
                  o => o.Id,
                  p => p.IdOsoby,
                  (osoba, pojazdy) =>
                     new {
                           osoba,
                           pojazdy = pojazdy.DefaultIfEmpty()
                         }
                  )
        .SelectMany(para => para.pojazdy,
                    (osoba, pojazd) =>
                      new { osoba.Imie,
                           osoba.Nazwisko,
                           NrRej = pojazd?.NrRej ?? "Brak pojazdu"
                         }
```

```
                    );
```

10.2 Opcje w C#

W przeciwieństwie do F# w C# nie istnieje wbudowany typ pozwalający na opisanie wartości opcjonalnych jednak możemy go łatwo zaimplementować np. w postaci klasy:

```
class Opcja<T>
{
  public bool MaWartosc { get; }
  public T Wartosc { get; }

  public Opcja()
  {
    MaWartosc = false;
  }

  public Opcja(T wartosc)
  {
    if (wartosc != null)
    {
      Wartosc = wartosc;
      MaWartosc = true;
    }
    else
      MaWartosc = false;
  }

  public Opcja<S> Mapuj<S>(Func<T,S> mapa)
  {
    switch(MaWartosc)
    {
      case true: return new Opcja<S>(mapa(Wartosc));
      case false: return new Opcja<S>();
    }
  }

  public Opcja<S> Zlacz<S>(Func<T, Opcja<S>> mapa)
  {
    switch (MaWartosc)
    {
      case true: return mapa(Wartosc);
      case false: return new Opcja<S>();
    }
  }
}
```

Zwróćcie uwagę, że klasę tę wyposażyłem dodatkowo w metody Mapuj i Zlacz realizujące odpowiednio operacje mapowania i bindowania. Dzięki nim możliwe jest stosowanie na obiektach opcjonalnych zwykłych funkcji podobnie jak to robiliśmy w

języku F# .

Oczywiście typ ten można wykorzystać zawsze, jeżeli mamy do czynienia z wartościami opcjonalnymi. Zmodyfikujmy poprzedni przykład, tak że osoba z pojazdem, będzie połączona nie za pomocą identyfikatora, tylko numeru pesel osoby. Dodatkowo numer ten w klasie osoba, będzie opcjonalny:

```
class Osoba
{
  public int Id { get; init; }
  public string Imie { get; init; }
  public string Nazwisko { get; init; }
  public Opcja<string> Pesel { get; init; }
}

class Pojazd
{
  public string NrRej { get; init; }
  public string Pesel { get; init; }
}
```

W metodzie Main możemy teraz utworzyć instancje tej klasy:

```
class Program
{
  static void Main(string[] args)
  {
    var osoba1 =
           new Osoba { Id = 1,
                       Imie="Ala",
                       Nazwisko = "Kot",
                       Pesel = new Opcja<string>("12345678901")
                     };
    var osoba2 =
           new Osoba { Id = 4,
                       Imie="Tomasz",
                       Nazwisko="Nowak",
                       Pesel = new Opcja<string>()
                     };
  }
}
```

Możemy teraz utworzyć funkcję, która będzie przyjmowała numer pesel i zwracała wszystkie pojazdy należące do tej osoby:

```
public IEnumerable<Pojazd> ZwrocPojazdy(string pesel)
{
  var pojazdy = new[]
  {
    new Pojazd {Pesel = "12345678901", NrRej="SC12345"},
    new Pojazd {Pesel = "23456789012", NrRej="WE12345"},
    new Pojazd {Pesel = "12345678901", NrRej="KW12345"}
  };

  return pojazdy.Where(pojazd => pojazd.Pesel == pesel);
}
```

Ponieważ funkcja ta przyjmuje wartość typu string jako swój argument wobec tego nie możemy wywołać jej bezpośrednio w następujący sposób:

```
var pojazdy = ZwrocPojazdy(osoba1.Pesel);
```

ponieważ mamy tu ewidentną niezgodność typów. Aby pobrać pojazdy w poprawny sposób powinniśmy wykorzystać funkcję Mapuj:

```
var pojazdy = osoba1.Pesel.Mapuj(ZwrocPojazdy);
```

Oczywiście zmienna pojazdy będzie miała teraz typ Opcja<IEnumerable<Pojazd». Jednak ma to oczywiście sens ponieważ skoro pesel jest opcjonalny, to również kolekcja pojazdów pobranych na jej podstawie powinna być opcjonalna.

Rozszerzmy nasz problem jeszcze raz. Dodajmy teraz do klasy Osoba również właściwość Kierownik typu Opcja<Osoba>:

```
class Osoba
{
  public int Id { get; init; }
  public string Imie { get; init; }
  public string Nazwisko { get; init; }
  public Opcja<string> Pesel { get; init; }
  public Opcja<Osoba> Kierownik {get; init;}
}
```

Oczywiście właściwość Kierownik powinna być opcjonalna, ponieważ nie każdy ma nad sobą kierownika.

Załóżmy teraz, że będziemy chcieli pobrać wszystkie pojazdy należące do kierownika danej osoby. Jeżeli zrealizujemy to w następujący sposób:

```
var pojazdyKierownika =
          osoba1
         .Kierownik
         .Mapuj(kierownik=>kierownik.Pesel.Mapuj(ZwrocPojazdy));
```

Pomyślcie chwilę o typie jaki będzie przypisany do zmiennej pojazdyKierownika. Dojdziecie do podobnej konkluzji jaka już pojawiła się podczas programowania funkcyjnego, że będzie to: Opcja<Opcja<IEnumerable<Pojazdy>>>. Oczywiście podwójna opcja będzie nam bardzo utrudniała życie, dlatego należałoby spłaszczyć tę hierarchię. Możemy to zrobić zamieniając pierwsze wywołanie funkcji Mapuj na Zlacz:

```
var pojazdyKierownika =
          osoba1
         .Kierownik
         .Zlacz(kierownik=>kierownik.Pesel.Mapuj(ZwrocPojazdy));
```

10.3 Opcje, a Linq

Jeżeli porównalibyśmy sygnatury metod Mapuj z opcji i Select dla IEnumerable, oraz Zlacz i SelectMany zauważymy, że są one bardzo podobne, a różnią się wyłącznie typem na którym działają. Ale co ważne ich ogólna idea działania jest dokładnie taka sama. W końcu już sobie wyjaśnialiśmy, że koncepcyjnie, to opcje i kolekcje policzalne są do siebie bardzo podobne. Skoro tak to jeżeli dla kolekcji typu IEnumerable możemy zapisać następujące zapytanie:

```
var kolekcja = Enumerable.Range(0,100);
var wartosci = from x in kolekcja
               select Math.Sin(x);
```

to może dla naszego typu Opcja moglibyśmy zapisać następujący kod:

```
var opcja = new Opcja<int>(12);
var wartosci = from x in opcja
               select Math.Sin(x);
```

Okazuje się, że jest to możliwe, tylko musimy stworzyć odpowiednie metody rozszerzające. Przykładowo, aby zapisać powyższe zapytanie powinniśmy stworzyć metodę rozszerzającą Select:

```
static class Opcja
{
  public static Opcja<S> Select<T, S>(
      this Opcja<T> opcja,
      Func<T, S> mapa)
    => opcja.Mapuj(mapa);
}
```

Dzięki temu przykład, w którym pobieraliśmy dla osoby możemy teraz zapisać następująco:

```
var pojazdy = from pesel in osoba1.Pesel
              select ZwrocPojazdy(pesel);
```

Podobnie jeżeli napiszemy dodatkowo metodę SelectMany w następującej postaci:

```
static class Opcja
{
   public static Opcja<S> SelectMany<T, R, S>(
   this Opcja<T> opcja,
   Func<T, Opcja<R>> mapa,
   Func<T, R, S> mapaWyniku
   ) =>
    opcja.Zlacz(
      wartosc => mapa(wartosc).Zlacz(
      wynik => new Opcja<S>(mapaWyniku(wartosc, wynik))));
}
```

zapytanie o pojazdy kierownika, będziemy mogli zapisać w formie:

```
var pojazdyKierownika = from kierownik in osoba1.Kierownik
                        from pesel in kierownik.Pesel
                        select ZwrocPojazdy(Pesel);
```

Podobnie moglibyśmy napisać odpowiednie funkcje dla pozostałych klauzul zapytań Linq.

10.4 Zadania do samodzielnego wykonania

Zadanie 10.1 Napisz funkcję, w której wykorzystasz Linq do utworzenia n-elementowej tablicy całkowitych liczb losowych o wartościach z określonego przedziału.

Zadanie 10.2 Napisz zapytanie Linq, które pobierze z tablicy utworzonej w poprzednim zadaniu m największych wartości.

Zadanie 10.3 Napisz zapytanie Linq, które określi sumę elementów w tablicy utworzonej w zadaniu 1.

Zadanie 10.4 Napisz zapytanie Linq, które określi średnią wartość elementów w tablicy utworzonej w zadaniu 1.

Zadanie 10.5 Napisz zapytanie Linq, które określi histogram tablicy utworzonej w zadaniu 1.

Zadanie 10.6 Napisz zapytanie Linq, które określi histogram wartości parzystych w tablicy utworzonej w zadaniu 1.

Zadanie 10.7 Napisz zapytanie Linq, które określi jaka wartości występowała najczęściej w tablicy z zadania 1.

Zadanie 10.8 Napisz zapytanie Linq, które z tablicy wyrazów wybierze te o długości większej niż 6 i zwróci je zapisane małymi literami.

Zadanie 10.9 Napisz program, w którym policzysz iloczyn skalarny dwóch tablic.

Zadanie 10.10 Napisz funkcję rozszerzającą możliwości Linq, która przjmie dwie kolekcje policzalne i wróci kolekcję wszystkich możliwych par, jakie można utworzyć z tych elementów.

Zadanie 10.11 Napisz program, który wczyta plik tekstowy linia po linii, a następnie policzy ile różnych wyrazów wystąpiło w tym pliku.

Zadanie 10.12 Napisz program, który wczyta plik tekstowy linia po linii, a następnie policzy ile było w nim wyrazów jednoliterowych, dwuliterowych, trzyliterowych itd.

Zadanie 10.13 Napisz program, który wczyta plik tekstowy linia po linii, a następnie obliczy ile razy wystąpił każdy znak w tym pliku. Wyniki wyświetl w kolejności rosnącej wg znaku.

Zadanie 10.14 Stwórz klasy na podstawie pokazanych przykładów:

```
var produkty = new [] {
  new Produkt {Id = 1, Nazwa = "Dysk HDD", Cena=500},
  new Produkt {Id = 2, Nazwa = "Monitor", Cena=1000},
  new Produkt {Id = 3, Nazwa = "Obudowa", Cena=300},
  new Produkt {Id = 4, Nazwa = "Plyta glowna", Cena=400},
  new Produkt {Id = 5, Nazwa = "Procesor", Cena=800}
};

var pozycje = new [] {
  new Pozycja {Id=1, IdProduktu=1, Liczba=2}
  new Pozycja {Id=2, IdProduktu=2, Liczba=3}
  new Pozycja {Id=3, IdProduktu=3, Liczba=1}
  new Pozycja {Id=4, IdProduktu=2, Liczba=2}
  new Pozycja {Id=5, IdProduktu=2, Liczba=1}
  new Pozycja {Id=6, IdProduktu=1, Liczba=3}
  new Pozycja {Id=7, IdProduktu=2, Liczba=3}
}
```

Na tej podstawie napisz zapytanie Linq, które określi:

1. cenę najdroższego produktu,
2. liczbę produktów w cenie powyżej 400 zł,
3. średnią cenę produktów w cenie poniżej 500 zł,
4. łączną cenę wszystkich produktów,
5. łączna cenę produktów, których nazwa zaczyna się na literę 'P',
6. nazwę produktu oraz liczbę sztuk produktów, które zostały sprzedane (tylko dla produktów, które zostały sprzedane),
7. identyfikator produktu oraz łączną liczbę sprzedanych produktów o tym identyfikatorze,
8. nazwę produktu oraz łączną kwotę, którą pozyskano ze sprzedaży (tylko dla produktów, które zostały sprzedane),
9. nazwę produktu oraz liczbę sztuk produktów, które zostały sprzedane (dla wszystkich produktów),
10. nazwę produktu oraz łączną kwotę, którą pozyskano ze sprzedaży (dla wszystkich produktów)

Zadanie 10.15 Napisz funkcję, która rozszerzy możliwości Linq, tak aby możliwe było określenie najdroższego produktu. Funkcja powinna być generyczna pozwalając działać na dowolnych obiektach i przyjmować funkcję, która zwróci wartość właściwości dla której ma być określona wartość maksymalna.