

ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej

Aplikacje webowe na platformę .NET

W07 – Język C#: inne

Syllabus

- Gotowe klasy generyczne
- Idea delegata
- Delegat w C#
- Generyczne delegaty – `Action` i `Func`
- Lambdy
 - W postaci bloku instrukcji
 - W postaci wyrażenia
- Niuanse delegatów
- Technologia LINQ - uzasadnienie
- Składnia:
 - W postaci kwerendy
 - W postaci metod
- Rodzaje operatorów
- Przykłady użycia operatorów
- Opóźnione i natychmiastowe wykonywanie kwerend
- Kwerenda po kompilacji
- Mechanizm refleksji
 - Obiekty `Type`
 - Metody `GetMethod` i `Invoke`
 - Operator **`nameof`**
 - Klasy `Assembly` i inne
 - Dostęp do atrybutów z klasy `Type`

Kolekcje

- Istnieje wiele klas już zaimplementowanych kolekcji.
- Wszystkie implementują interfejs `IEnumerable<out T>` (który implementuje `IEnumerable`).
- Posiada tylko jedną metodę:

```
IEnumerator<out T> GetEnumerator ();
```
- Interfejs `IEnumerator<out T>` posiada
 - właściwość `Current` pobierający bieżący element kolekcji
 - Metodę `MoveNext()` do przejścia do kolejnego elementu
 - Metodę `Reset()` do przejścia przed początek kolekcji (tam powinien stać enumerator po stworzeniu)
 - Metodę `Dispose()`
- Dwie pierwsze metody pozwalają na używanie pętli `foreach(...)`.
 - Dla tej pętli i kolekcji używana jest też technika duck typing (kacze typowanie).
- Interfejs `IEnumerable` posiada ponad setkę metod rozszerzających, które zwracają kolekcję posortowaną, przefiltrowaną itd. Kolekcja taka implementuje interfejs `IEnumerable` zatem operacje na kolekcji można wyrazić jako ciąg (notacja z kropką) kolejnych wywołań metod rozszerzających.
- Przykładowe kolekcje:
 - `List<T>`
 - `Queue<T>`
 - `Dictionary<Tkey, TValue>`
 - `Stack<T>`
 - `SortedSet<T>`
 - `HashSet<T>`
- Istnieją kolekcje tylko do odczytu (`ReadOnly`), współbieżne (`Concurrent`), z niemutowalnymi elementami (`Immutable`) itd.
- Każda kolekcja ma również metody adekwatne do struktury, którą implementuje.

Kolekcje – przykłady działania

```
public static void QueueProbe()  
{  
    Queue<string> queueS = new Queue<string>()  
    Console.WriteLine("Queue:");  
    queueS.Enqueue("one");  
    queueS.Enqueue("two");  
    queueS.Enqueue("three");  
    foreach (string s in queueS)  
        Console.WriteLine(s);  
    while (queueS.Count!=0)  
        Console.WriteLine(queueS.Dequeue());  
    Console.WriteLine("Queue after dequeu:");  
    foreach (string s in queueS)  
        Console.WriteLine(s);  
}
```

```
Queue:  
one  
two  
three  
one  
two  
three  
Queue after dequeu:
```

```
public static void DictionaryProbe()  
{  
    var dict = new Dictionary<int, string>();  
    dict.Add(5, "Aleksy");  
    dict.Add(3, "John");  
    dict.Add(8, "Proxy");  
    dict.Add(4, "cat");  
    Console.WriteLine("dictionary:");  
    foreach (var elem in dict)  
        Console.WriteLine(elem);  
    Console.WriteLine("keys:");  
    foreach (var elem in dict.Keys)  
        Console.WriteLine(elem);  
    Console.WriteLine("access to one elem by a key (very fast):");  
    Console.WriteLine(dict.GetValueOrDefault(8));  
    Console.WriteLine(dict.GetValueOrDefault(1));  
    Console.WriteLine("Remove() returns:" + dict.Remove(5));  
    Console.WriteLine("TryAdd() returns:" + dict.TryAdd(8, "nonproxy"));  
}
```

```
dictionary:  
[5, Aleksy]  
[3, John]  
[8, Proxy]  
[4, cat]  
keys:  
5  
3  
8  
4  
access to one elem by a key (very fast):  
Proxy  
  
Remove() returns:True  
TryAdd() returns:False
```

Inicjatory kolekcji

- Zamiast jawnie wywoływać operację `Add()` na kolekcji można użyć inicjatorów kolekcji
- Składnia przypomina inicjatory obiektów lub tablic.
- Kompilator przerobi taki kod jak na wywołanie kolejnych metod `Add()`.

```
public static void Print<T>(IEnumerable<T> items)
{
    foreach (T item in items)
        Console.WriteLine(item);
}

public static void InitiatorStringCollection()
{
    List<string> dayOfWeek = new List<string>()
    {
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        "Sunday",
    };
    Print(dayOfWeek);
}
```

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

Inicjatory kolekcji słownikowej

- W przypadku kolekcji, która jest słownikiem możliwe jest używanie notacji tablicowej oraz znaku podstawienia (od C# 6.0)

```
public static void InitiatorDictionary()
{
    // before C# 6.0
    Dictionary<string, int> personAgeMapOld = new Dictionary<string, int>()
    {
        { "John" , 23 },
        { "Alice", 30 },
        { "Fred", 50 }
    };
    // from C# 6.0
    Dictionary<string, int> personAgeMap = new Dictionary<string, int>()
    {
        ["John"] = 23,
        ["Alice"] = 30,
        ["Fred"] = 50
    };
    Print(personAgeMapOld);
    Console.WriteLine("-----");
    Print(personAgeMap);
}
```

```
[John, 23]
[Alice, 30]
[Fred, 50]
```

```
-----
[John, 23]
[Alice, 30]
[Fred, 50]
```

Informacje różne

- Kowariancja i kontrawariancja – problem, gdy działamy na kolekcjach elementów klasy A i B które są połączone dziedziczeniem:
 - Kurs „Programowanie funkcyjne”
- Pisanie własnych klas generycznych jest dość trudną sprawą, gdyż użytkownik będzie chciał jej użyć w każdym możliwym kontekście, zatem trzeba zaimplementować metody do porównań, szybkie metody do kopiowania itp. (oraz do poprawnego działania współbieżnego)
- Pisanie własnych kolekcji generycznych to jeszcze wyższe wymagania (tworzenie generycznych iteratorów, operatorów porównujących itd.)
- Dobrym testem jest użycie zaimplementowanych klas generycznych w tych samych klasach generycznych lub systemowych kolekcjach generycznych (czyli tworzenie kolekcji kolekcji)

C#

DELEGATY I LAMBDY

Idea delegata

- Często, w przypadku pracy na kolekcjach (ale nie tylko) i tworzenia ogólnych algorytmów potrzeba podczas wykonania podać jak wykonać określoną operację (np. algorytm sortowania – porównanie elementów). Dodatkowo ta operacja może się zmieniać w kolejnym wykonaniu tego samego algorytmu (raz sortujemy wg nazwiska, raz wg wieku innym razem łączymy jakoś te dwie cechy itp.). Możliwe rozwiązania:
 - Tworzymy instrukcję **switch** i w zależności od parametru (kodu oznaczającego np. co porównujemy) wykonujemy inny fragment kodu
 - Wada: każdy nowy sposób porównywania elementów – trzeba dopisać kod w metodzie sortującej, skompilować itd.
 - Tworzymy interfejs zawierający tylko jedną metodę porównującą i staje się on parametrem metody sortującej (Java)
 - Musimy stworzyć obiekt, aby z tego skorzystać
 - Tworzymy nagłówek metody delegata, którego potem używamy w nagłówku metody sortującej. Podczas wywołania w to miejsce wstawiamy nazwę metody statycznej (zgadzającej się co do nagłówka)

Delegat

- Delegat to specjalny typ referencyjny.
- Nie jest wprost klasą.
- Dziedziczy po klasie `System.Delegate`
 - właściwie w .Net po klasie `System.MulticastDelegate`, która dziedziczy po klasie `System.Delegate`
- Jednak nie piszę się tego dziedziczenia.
- Składnia deklaracji delegata wygląda jak składnia metody abstrakcyjnej, tylko posiada słowo kluczowe **delegate**.
- Może być przekazywany jako parametr metody.
- Może być zapamiętany w zmiennej referencyjnej (w specjalny sposób), jako pole itp.
 - Jako typu generycznego `Action` lub `Func`

Delegat – przykład – 1/2

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
    public override string ToString()
    {
        return $"{Name}, {Age}";
    }
}
```

```
class Program
{
    public delegate bool LessThan(Person a, Person b);

    public static void BubbleSort(Person[] tab, LessThan lessThan)
    {
        for(int i=tab.Length-1;i>0 ;i--)
            for (int j = 0; j < i; j++)
                if (!lessThan(tab[j], tab[j + 1]))
                {
                    Person p = tab[j];
                    tab[j] = tab[j + 1];
                    tab[j + 1] = p;
                }
    }
}
```

Delegat – przykład – 2/2

```
public static bool LessName(Person x, Person y)
{
    return x.Name.CompareTo(y.Name)<0;
}
public static bool LessAge(Person x, Person y)
{
    return x.Age < y.Age;
}
public static void showPerson(string comment, Person[] tab)
{
    Console.WriteLine(comment);
    foreach (Person p in tab)
        Console.WriteLine(p);
}
public static void BubbleSortTest()
{
    Person[] tab = { new Person("Nowak", 45), new Person("Nowak", 30),
                    new Person("Kowal", 40), new Person("Wojtas", 30) };
    BubbleSort(tab, LessName);
    showPerson("wg imienia:", tab);
    BubbleSort(tab, LessAge);
    showPerson("wg wieku:", tab);
}
```

```
by name:
Kowal, 40
Nowak, 45
Nowak, 30
Wojtas, 30
by age:
Wojtas, 30
Nowak, 30
Kowal, 40
Nowak, 45
```

Generyczne delegaty – Action i Func

- W języku C# 4.0 wprowadzono generyczne deklaracje delegatów. Funkcje nie zwracające wartości mają nazwę `Action`, a te które zwracają wartość nazwę `Func` (ostatni parametr typu generycznego jest typem zwracanej wartości):
 - `public delegate void Action();`
 - `public delegate void Action<in T>(T arg);`
 - ...
 - `public delegate void Action<in T1, in T2, in T3>(T1 arg1, T2 arg2, T3 arg3);`
 - ...
 - `public delegate TResult Func<out TResult>>();`
 - `public delegate TResult Func<in T, out TResult>(T arg);`
 - ...
 - `public delegate TResult Func<in T1, in T2, in T3, out TResult>(T1 arg1, T2 arg2, T3 arg3);`
 - ...
- Stąd metodę do sortowania można również zadeklarować bez tworzenia własnego typu delegata `ComparisonHandler`:
 - `void BubbleSort(Person[] item, Func<Person, Person, bool> lessThan) {...}`

Lambdy (w postaci bloku instrukcji)

- Zamiast tworzyć metodę statyczną z nadaną nazwą, można stworzyć funkcję anonimową (nie zalecane, nie będzie tu omawiane)
- Zamiast metod anonimowych zalecane jest używanie **wyrażeń lambda**.
- Wyrażenie lambda można zapisać:
 - W postaci instrukcji
 - W postaci wyrażenia
- Kompilator przekształca (choć nie zawsze) wyrażenie lambda na delegata, chociaż nie podawana jest nazwa delegata.
- Składnia:
 - `<Lista parametrów formalnych> => <blok kodu>`
- Lista parametrów formalnych jest w nawiasach okrągłych, chyba, że jest tylko jeden parametr bez podania jego typu.
- Blok kodu znajduje się między klamrami (jak ciało metody)
- W lambdzie nie podaje się typu zwracanego, kompilator sam go wywnioskuje.
- Podobnie można nie podawać typów argumentów.

```
BubbleSort(tab, (Person x, Person y)=> { return x.Name.CompareTo(y.Name) < 0; });  
showPerson("wg imienia:", tab);  
BubbleSort(tab, (x, y) => { return x.Age < y.Age; });  
showPerson("wg wieku:", tab);
```

Lambda w postaci wyrażenia

- Lambdy w postaci wyrażenia są jeszcze bardziej zwięzłą formą pisania lambd.
- Składnia:
 - `<Lista parametrów formalnych> => <wyrażenie>`
- Wyrażenie jest bez żadnych nawiasów, czy klamr
- Ograniczenie: wyrażenie to nie ciąg instrukcji, wynik trzeba zapisać jako wyrażenie.
- Zysk: nie trzeba pisać również słowa `return`.

```
BubbleSort(tab, (Person x, Person y) => x.Name.CompareTo(y.Name) < 0);  
showPerson("wg imienia:", tab);  
BubbleSort(tab, (x, y) => x.Age < y.Age);  
showPerson("wg wieku:", tab);
```

Lambda w pętli for 1/3

```
static void testForX()
{
    var items = new string[] { "Bolek", "Lolek", "Tola" };
    var actions = new List<Action>();
    for (int i = 0; i < items.Length; i++)
    {
        actions.Add(() => { Console.WriteLine(items[i]); });
    }
    foreach (Action action in actions)
        action();
}
```

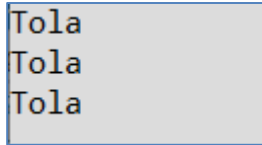
- Zmienna `i` ma wartość 3 w momencie uruchomienia lambdy

```
Unhandled exception. System.IndexOutOfRangeException: Index was outside the bounds of the array.
   at DelegatesLambdas.Program.<>c__DisplayClass11_1.<testForX>b__0() in C:\Users\dariu\source\repos\DelegatesLambdas\DelegatesLambdas\Program.cs:line 109
   at DelegatesLambdas.Program.testForX() in C:\Users\dariu\source\repos\DelegatesLambdas\DelegatesLambdas\Program.cs:line 112
   at DelegatesLambdas.Program.Main() in C:\Users\dariu\source\repos\DelegatesLambdas\DelegatesLambdas\Program.cs:line 19
```


Lambda w pętli for 2/3

```
static void testFor()
{
    var items = new string[] { "Bolek", "Lolek", "Tola" };
    var actions = new List<Action>();
    string item;
    for (int i = 0; i < items.Length; i++)
    {
        item = items[i];
        actions.Add(() => { Console.WriteLine(item); });
    }
    foreach (Action action in actions)
        action();
}
```

- Wszystkie lambdy używają jednej zmiennej `item`, której ostatnie poprawne przypisanie to `item=items[2]`.



Tola
Tola
Tola

Lambda w pętli for 3/3

```
static void testForIn()
{
    var items = new string[] { "Bolek", "Lolek", "Tola" };
    var actions = new List<Action>();
    for (int i = 0; i < items.Length; i++)
    {
        string item;
        item = items[i];
        actions.Add(() => { Console.WriteLine(item); });
    }
    foreach (Action action in actions)
        action();
}
```

- Każda lambda ma swoją zmienną `item` – oczekiwane działanie.

```
Bolek
Lolek
Tola
```

Lambda w pętli foreach

- To samo działanie osiągnięte za pomocą pętli **foreach**.

```
static void testForeach()
{
    var items = new string[] { "Bolek", "Lolek", "Tola" };
    var actions = new List<Action>();
    foreach (string item in items)
        actions.Add(() => { Console.WriteLine(item); });
    foreach (Action action in actions)
        action();
}
```

```
Bolek
Lolek
Tola
```

Parametr – kopia referencji

```
class Counter {  
    public int InnerCounter { get; set; }  
}  
  
static Action CreateLamda(Counter x)  
{  
    Action action;  
    action = () => { Console.WriteLine(x.InnerCounter); };  
    return action;  
}  
  
static void TestReturnLamda()  
{  
    Counter c = new Counter { InnerCounter = 5 };  
    Action ac1 = CreateLamda(c);  
    c.InnerCounter++;  
    Action ac2 = CreateLamda(c);  
    ac1();  
    ac2();  
}
```

6
6

- Ten sam problem co wcześniej, ale głębiej ukryty
– Rozwiązanie: klon obiektu przekazać do lambdy

Wyrażenia Lambda

- Wyrażenia lambda nie zawsze są kompilowane do kodu wykonywalnego.
- Często są wykorzystywane w mechanizmie odbicia do:
 - Stworzenia drzewa wyrażenia
 - Przerobienia kwerendy w języku LINQ do kwerendy w języku SQL i wysłania kwerendy na serwer SQL
 - Analizy adnotacji:

```
<dt>  
    @Html.DisplayNameFor(model => model.Nazwisko )  
</dt>
```

- Lambdy muszą korzystać albo z parametrów, albo ze zmiennych, które będą istnieć w trakcie wykonywania lambda.
 - Nie można używać parametrów przekazanych przez **out**, **ref**, **in** w ciele lambda

C#

JĘZYK LINQ

Działania na zestawach danych

- Najczęstsze operacje na zestawach danych to:
 - Filtrowanie kolekcji (odrzućenie niepotrzebnych elementów)
 - Projekcja (przekształcenie elementów na inną postać)
 - Łączenie kolekcji
- Kolekcje danych mogą być zapamiętane w **różnych strukturach** danych (tablice, listy, słowniki itd.)
- Dane mogą być zarówno wprost w pamięci w zmiennych, jak również w bazach danych, plikach, strumieniach itd.
 - Oczywiście istnieją obiekty do operacji na konkretnych kolekcjach
- Wskazane byłoby posiadanie języka zapytań, który pozwoli operować na kolekcjach danych **niezależnie** od ich struktury jak i rzeczywistego miejsca przechowywania.
- Jednym z rozwiązań jest język zapytań LINQ (Language-Integrated Query) definiujący dodatkowo **język zapytań z kwerendami** zbliżony składnią do języka SQL.
 - Method Base Syntax (MBS)
 - Użyte jest tu Fluent API, czyli wynikiem operacji jest kolekcja, więc ponownie można użyć notacji z kropką dla kolejnej operacji.
 - Query Expression Syntax (QES)
 - Pojawiają się tu **kontekstowe** słowa kluczowe.

Przykład kwerendy LINQ

- Należy zaimportować `System.Linq`

```
static void FirstLINQ()
{
    string[] words = { "ala", "monkey", "cat", "dolly", "wind", "water" };
    IEnumerable<string> selection =
        from sequence in words
        where sequence.Contains("a")
        select sequence;
    foreach (string word in selection)
    {
        Console.WriteLine(word);
    }
}
```

```
ala
cat
water
```


Składnia LINQ

- Składnia wyrażenia QES z kwerendą jest zbliżona do wyrażień z języka SQL.
- Jednak zmieniona została kolejność składowych np. **from...in... where... select...**
 - Pozwala to na działanie IntelliSense
 - Jest to ogólnie bardziej logiczne
 - Zmienne zakresowe najpierw są określane, a dopiero potem używane
- Wynik to najczęściej kolekcja typu `IEnumerable<T>` lub `IQueryable<T>`
 - Operatory agregujące mogą zwracać inne typy
- Typ T jest określany na podstawie klauzuli **select** lub **group by**.

LINQ - przekształcenie

- W rzeczywistości kod zostanie przekształcony na zapytanie LINQ w postaci zapisu Fluent API (MBS)
 - Czyli dla wyniku zapytania QES można wykonać kolejne zapytanie za pomocą notacji z kropką. I odwrotnie.

```
static void FirstLINQFluent()  
{  
    string[] words = { "ala", "monkey", "cat", "dolly", "wind", "water" };  
    IEnumerable<string> selection = words  
        .Select(s => s)  
        .Where(s => s.Contains("a"))  
        .Select(s => s);  
    selection.ToList().ForEach(Console.WriteLine);  
}
```

ala
cat
water

- Czy wręcz w tym przypadku po uproszczeniu


```
static void FirstLINQFluentSimpler()  
{  
    string[] words = { "ala", "monkey", "cat", "dolly", "wind", "water" };  
    IEnumerable<string> selection = words  
        .Where(s => s.Contains("a"));  
    selection.ToList().ForEach(Console.WriteLine);  
}
```

ala
cat
water

LINQ- informacje

- Język ten można bardzo łatwo wykorzystać do komunikowania się z bazą danych, pod warunkiem istnienia klas w C# mapujących tabele.
 - Entity Framework jest biblioteką do tego przygotowaną.
- Większość operacji wykorzystywanych w LINQ na kolekcjach jest zrealizowana jako metody rozszerzające do interfejsu `IEnumerable<T>`:
 - <https://docs.microsoft.com/pl-pl/dotnet/api/system.collections.generic.ienumerable-1?view=netcore-3.1>
 - <https://docs.microsoft.com/pl-pl/dotnet/api/system.collections.generic.ienumerable-1?view=net-5.0>
- Zapis w postaci kwerendy (czyli zbliżony do języka SQL) zawsze można przekształcić na zapis w postaci Fluent API, jednak nie odwrotnie.
 - Zapis w postaci kwerendy jest często prostszy.
- W celu użycia LINQ do operacji na bazie danych można posłużyć się narzędziem LINQPad:

– <https://www.linqpad.net/>

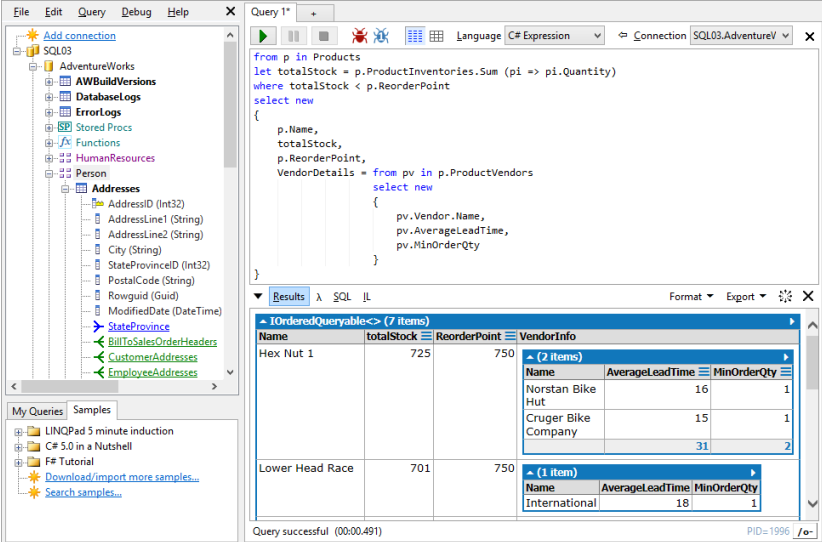


The .NET Programmer's Playground

- Instantly test any C#/F#/VB snippet or program
- Query databases in LINQ (or SQL) — SQL/Azure, Oracle, SQLite, Postgres & MySQL
- Enjoy rich output formatting, optional autocompletion and integrated debugging
- Script and automate in your favorite .NET language
- Super lightweight — single 20MB executable!
- Standard edition free, with no expiry

3 million downloads

Download LINQPad



The screenshot shows the LINQPad application window. On the left is a file explorer showing a project named 'AdventureWorks' with various folders like 'AdventureWorks', 'AWBuildVersions', 'DatabaseLogs', 'ErrorLogs', 'Stored Procs', 'Functions', 'HumanResources', 'Person', 'Addresses', 'StateProvince', 'SalesOrderHeaders', 'CustomerAddresses', and 'EmployeeAddresses'. The main editor displays a LINQ query:

```
from p in Products
let totalStock = p.ProductInventories.Sum(pi => pi.Quantity)
where totalStock < p.ReorderPoint
select new
{
    p.Name,
    totalStock,
    p.ReorderPoint,
    VendorDetails = from pv in p.ProductVendors
                    select new
                    {
                        pv.VendorName,
                        pv.AverageLeadTime,
                        pv.MinOrderQty
                    }
}
```

Below the query, the 'Results' tab shows a table with 7 items. The table has columns: Name, totalStock, ReorderPoint, and VendorInfo. The data is as follows:

Name	totalStock	ReorderPoint	VendorInfo
Hex Nut 1	725	750	(2 items)
			Name AverageLeadTime MinOrderQty
			Norstan Bike Hut 16 1
			Cruiger Bike Company 15 1
			31 2
Lower Head Race	701	750	(1 item)
			Name AverageLeadTime MinOrderQty
			International 18 1

At the bottom, it says 'Query successful (00:00.491)' and 'PID=1996'.

Grupy operacji/operatorów w ramach LINQ

- Operatory, które można używać w ramach LINQ można podzielić na grupy:
 - Operatory agregacji (Aggregate operators)
 - Operatory grupowania (Grouping operators)
 - Operatory ograniczeń (Restriction operators)
 - Operatory projekcji (Projection operators)
 - Operatory zbiorowe (Set operators)
 - Operatory podziału (Partitioning operators)
 - Operatory konwersji (Conversion operators)
 - Operatory generacji (Generation operators) kolekcji
 - Operatory łączenia (Join Operators)
 - Operatory porównywania sekwencji (Custom sequence operators)
 - Operatory kwantyfikacji (Quantifiers Operators)
 - Inne operatory (Miscellaneous operators)
- W celach demonstracyjnych na kolejnych slajdach wykorzystywana będzie tablica liczb, tablica ciągów znaków, lista obiektów klasy `Student` oraz dostęp do bazy demonstracyjnej `DemoDb.sdf` tworzonej podczas instalacji LINQPad.

Używane klasy demonstracyjne

- Poniższe klasy mają również odpowiednie konstruktory i metodę ToString().

```
public class Department
{
    public int Id { get; set; }
    public String Name { get; set; }
    //...
}
public enum Gender
{
    Female, Male
}
public class StudentWithTopics
{
    public int Id { get; set; }
    public int Index { get; set; }
    public string Name { get; set; }
    public Gender Gender { get; set; }
    public bool Active { get; set; }
    public int DepartmentId { get; set; }

    public List<string> Topics { get; set; }
    //...
}
```

```
public static List<string> GenerateNamesEasy()
{
    return new List<string>() {
        "Nowak",
        "Kowalski",
        "Schmidt",
        "Newman",
        "Bandingo",
        "Miniwiliger"
    };
}
```

Używane kolekcje

```
public static void ShowAllCollections()  
{  
    Generator.GenerateIntsEasy().ToList().ForEach(Console.WriteLine);  
    Generator.GenerateDepartmentsEasy().ForEach(Console.WriteLine);  
    Generator.GenerateStudentsWithTopicsEasy().ForEach(Console.WriteLine);  
}
```

```
9  
7  
1  
2  
6  
7  
8  
1), Computer Science  
2), Electronics  
3), Mathematics  
4), Mechanics  
1) 12345,      Nowak, Female,  active, 1, topics: C#, PHP, algorithms,  
2) 13235,      Kowalski,  Male,   active, 1, topics: C#, C++, fuzzy logic,  
3) 13444,      Schmidt,   Male,no active, 2, topics: Basic, Java,  
4) 14000,      Newman, Female,no active, 3, topics: JavaScript, neural networks,  
5) 14001,      Bandingo,  Male,   active, 3, topics: Java, C#,  
6) 14100, Miniwiliger,  Male,   active, 2, topics: algorithms, web programming,  
11) 22345,      Nowak, Female,  active, 2, topics: C#, PHP, algorithms,  
12) 23235,      Nowak,   Male,   active, 1, topics: C#, C++, fuzzy logic,  
13) 23444,      Schmidt,  Male,no active, 1, topics: Basic, Java,  
14) 24000,      Newman, Female,no active, 1, topics: JavaScript, neural networks,  
15) 24001,      Bandingo,  Male,   active, 3, topics: Java, C#,  
16) 24100,      Bandingo,  Male,   active, 2, topics: algorithms, web programming,
```

Operatory filtrowania (ograniczeń)

- W przypadku kwerendy LINQ (QES) jest to klauzula **where** po której następuje logiczne wyrażenie filtrujące.
- W przypadku Fluent API LINQ (MBS) jest to metoda `Where` z delegatem/lambdą zwracającą wartość logiczną.

```
public static void MethodWhereSimple()
{
    var resInt = Generator.GenerateIntsEasy().Where(x => x % 2 == 0);
    resInt.ToList().ForEach(Console.WriteLine);
    var resStr = Generator.GenerateNamesEasy().Where(s => s.Length>6);
    resStr.ToList().ForEach(Console.WriteLine);
    var resStud = Generator.GenerateStudentsWithTopicsEasy()
        .Where(s => s.Active && s.DepartmentId==1);
    resStud.ToList().ForEach(Console.WriteLine);
}
```

```
2
6
8
Kowalski
Schmidt
Bandingo
Miniwiliger
1) 12345,      Nowak, Female,   active, 1, topics: C#, PHP, algorithms,
2) 13235,      Kowalski,   Male,     active, 1, topics: C#, C++, fuzzy logic,
12) 23235,     Nowak,     Male,     active, 1, topics: C#, C++, fuzzy logic,
```

Kwerendy LINQ (QES)

- Zapis dla tak prostego przykładu wygląda na dłuższy, jednak nie trzeba pisać tyle nawiasów i wprost wyrażen lambda.
- Efekt działa taki sam jak na poprzedniej stronie

```
public static void ClauseWhereSimple()
{
    var resInt = from v in Generator.GenerateIntsEasy()
                 where v % 2 == 0
                 select v;
    resInt.ToList().ForEach(Console.WriteLine);
    var resStr = from s in Generator.GenerateNamesEasy()
                 where s.Length > 6
                 select s;
    resStr.ToList().ForEach(Console.WriteLine);
    var resStud = from s in Generator.GenerateStudentsWithTopicsEasy()
                  where s.Active && s.DepartmentId == 1
                  select s;
    resStud.ToList().ForEach(Console.WriteLine);
}
```

```
2
6
8
Kowalski
Schmidt
Bandingo
Miniwiliger
1) 12345,      Nowak, Female,  active, 1, topics: C#, PHP, algorithms,
2) 13235,      Kowalski,  Male,   active, 1, topics: C#, C++, fuzzy logic,
12) 23235,     Nowak,   Male,   active, 1, topics: C#, C++, fuzzy logic,
```


Proste zapytanie w LINQPad

- Pod prawym przyciskiem myszki mamy tworzenie szkieletu zapytania zarówno jako MBS jak i QES.

The screenshot displays the LINQPad 6 interface. On the left, a tree view shows the 'DemoDB.sdf' file with 'Tables' and 'Categories' folders. The 'Customers' table is selected, and a context menu is open, showing various LINQ methods like 'Customers.Take(100)', 'Customers.Count()', and 'Customers.Where(c => ...)'. The main query editor shows the following code:

```
from e in Employees
where e.LastName.Length>7
select e
```

Below the query editor, the 'Results' tab is active, displaying a table of 6 items. The table has columns: EmployeeID, LastName, FirstName, Title, BirthDate, HireDate, Address, City, Region, and F. The data rows are:

EmployeeID	LastName	FirstName	Title	BirthDate	HireDate	Address	City	Region	F
3	Leverling	Janet	Sales Representative	1963-08-30 00:00:00	1991-02-27 00:00:00	722 Moss Bay Blvd.	Kirkland	WA	S
5	Buchanan	Steven	Sales Manager	1955-03-04 00:00:00	1992-09-13 00:00:00	14 Garrett Hill	London	null	S
8	Callahan	Laura	Inside Sales Coordinator	1958-01-09	1993-01-30	4726 - 11th Ave. N.F.	Seattle	WA	S

Proste operatory agregacji

- Istnieje 5 wbudowanych prostych operatorów agregacji
 - Min
 - Max
 - Sum
 - Count
 - Average
- Operator `Min`, `Max` w wersji bezparametrycznej działają dla typów implementujących interfejs `Comparable<T>`
- Istnieją wersje z lambdą, która musi zwracać wartość liczbową (**int, long, float, double**). Ze zwracanych wartości zostanie wybrana wartość minimalna lub maksymalna
- Dla operatorów `Sum`, `Average` istnieją również wersję z lambda zwracającą wartość liczbową.
- Użycie z lambdą to jakby ukryte użycie operatora `Select`, np. dla ostatniego przykładu poniżej:
- `var resStrMin2 = strs.Select(s=>s.Length).Min();`
- Operatory te nie istnieją dla kwerend.

```
public static void SimpleAggregates()
{
    var ints = Generator.GenerateIntsEasy();
    var resMin = ints.Where(x => x % 2 == 0).Min();
    Console.WriteLine(resMin);
    var resMax = ints.Where(x => x % 2 == 0).Max();
    Console.WriteLine(resMax);
    var strs = Generator.GenerateNamesEasy();
    var resStrMin1 = strs.Min(); //dictionaryOrder
    Console.WriteLine(resStrMin1);
    var resStrMin2 = strs.Min(s=>s.Length); // minimum length
    Console.WriteLine(resStrMin2);
}
```

2
2
Bandingo
5

Operator Aggregate

- Operator `Aggregate` pozwala na rozbudowaną operację połączenia kolejnych elementów w jeden wynik składając go od lewej do prawej (fold left).
 - $f(\dots f(f(f(a_1, a_2), a_3), a_4) \dots, a_n)$
- Są 3 postacie (w uproszczeniu poniżej):
 - `X Agregate (Func<X, X, X>)`
 - Dla pustej kolekcji będzie wyjątek
 - Dla jednoelementowej tenże element będzie wynikiem
 - `Y Agregate (Y accum, Func<Y, X, Y>)`
 - $f(\dots f(f(f((acc, a_1), a_2), a_3), a_4) \dots, a_n)$
 - `Z Agregate (Y accum, Func<Y, X, Y>, Func<Y, Z>)`
 - Po wykonaniu akumulacji przekształcenie wyniku do oczekiwanej postaci

Operator Aggregate - przykład

```
public static void ComplexAggregates()
{
    var strs = Generator.GenerateNamesEasy();
    var strs0 = new string[] { };
    var strs1 = new string[] { "one" };
    Console.WriteLine("----- first form, one argument: lambda >>>>>>>");
    Console.WriteLine(strs.Aggregate((a, b) => a + "," + b));
    try{ Console.WriteLine(strs0.Aggregate((a, b) => a + "," + b)); }
    catch (InvalidOperationException e) { Console.WriteLine(e.Message); }
    Console.WriteLine(strs1.Aggregate((a, b) => a + "," + b));
    Console.WriteLine("----- second form, two arguments: accumulator, lambda");
    Console.WriteLine(strs.Aggregate("", (a, b) => a + "," + b));
    Console.WriteLine(strs0.Aggregate("", (a, b) => a + "," + b));
    Console.WriteLine(strs1.Aggregate("", (a, b) => a + "," + b));
    Console.WriteLine("----- third form, three arguments: accumulator, lambda, finish lambda");
    Console.WriteLine(strs.Aggregate("", (a, b) => a + "," + b, res=>res.Length));
    Console.WriteLine(strs0.Aggregate("", (a, b) => a + "," + b, res => res.Length));
    Console.WriteLine(strs1.Aggregate("", (a, b) => a + "," + b, res => res.Length));
    Console.WriteLine("----- finding average lenght in a complex way");
    var resStr = strs.Aggregate((0, ""),
        (tuple, str) => (tuple.Item1 + 1, tuple.Item2 + str), res=>((double)res.Item2.Length)/res.Item1);
    Console.WriteLine(resStr);
    //var avrLen= strs.Average(s => s.Length);
    //Console.WriteLine(avrLen);
}
```

```
----- first form, one argument: lambda >>>>>>>
Nowak,Kowalski,Schmidt,Newman,Bandingo,Miniwiliger
Sequence contains no elements
one
----- second form, two arguments: accumulator, lambda
,Nowak,Kowalski,Schmidt,Newman,Bandingo,Miniwiliger
,one
----- third form, three arguments: accumulator, lambda, finish lambda
51
0
4
----- finding average lenght in a complex way
7,5
```

Operator filtrowania Where dwuargumentowy

- Prócz „zwykłego” `Where()` istnieje również dwuparametryczne, które otrzymuje pozycję elementu w kolekcji.

```
public static void WhereWithPos()
{
    var resStr = Generator.GenerateNamesEasy()
        .Where((s, pos) => pos % 2 == 0);
    resStr.ToList().ForEach(Console.WriteLine);
    var resStud = Generator.GenerateStudentsWithTopicsEasy()
        .Where((s, pos) => s.Active && pos % 3 == 0);
    resStud.ToList().ForEach(Console.WriteLine);
    Console.WriteLine("-----");
    var resStud2 = Generator.GenerateStudentsWithTopicsEasy()
        .Where(s => s.Active)
        .Where((s, pos) => pos % 3 == 0);
    resStud2.ToList().ForEach(Console.WriteLine);
}
```

```
Nowak
Schmidt
Bandingo
1) 12345,      Nowak, Female,   active, 1, topics: C#, PHP, algorithms,
11) 22345,     Nowak, Female,   active, 2, topics: C#, PHP, algorithms,
-----
1) 12345,      Nowak, Female,   active, 1, topics: C#, PHP, algorithms,
6) 14100, Miniwiliger, Male,   active, 2, topics: algorithms, web programming,
15) 24001,     Bandingo, Male,    active, 3, topics: Java, C#,
```

Operatory projekcji - Select

- Istnieją dwa operatory projekcji:
 - Select
 - SelectMany
- W przypadku operatora `Select` z danych jednego elementu kolekcji możemy stworzyć nowy typ (nazwany, anonimowy lub krotkę).
 - Wraz z przekształceniami wartości
- Istnieje `Select`, gdzie lambda ma drugi argument będący numerem elementu w kolekcji (analogicznie jak dla operatora `Where`).

```
public static void TestSelect()
{
    var resStud = Generator.GenerateStudentsWithTopicsEasy()
        .Where(s => s.Index < 20000)
        .Select(s => new { Header=s.Id + " " + s.Index, s.Name });
    foreach(var x in resStud)
    {
        Console.WriteLine($" {x.Header} =====> {x.Name}");
    }
    Console.WriteLine("-----");
    var resStud2 = from s in Generator.GenerateStudentsWithTopicsEasy()
        where s.Index < 20000
        select new { Header = s.Id + " " + s.Index, s.Name };
    foreach (var x in resStud2)
    {
        Console.WriteLine($" {x.Header} =====> {x.Name}");
    }
    Console.WriteLine("-----");
    var resStud3 = from s in Generator.GenerateStudentsWithTopicsEasy()
        where s.Index < 20000
        select (Header : s.Id + " " + s.Index, s.Name);
    foreach (var x in resStud3)
    {
        Console.WriteLine($" {x.Header} =====> {x.Name}");
    }
}
```

```
1) 12345 =====> Nowak
2) 13235 =====> Kowalski
3) 13444 =====> Schmidt
4) 14000 =====> Newman
5) 14001 =====> Bandingo
6) 14100 =====> Miniwiliger
```

```
-----
1) 12345 =====> Nowak
2) 13235 =====> Kowalski
3) 13444 =====> Schmidt
4) 14000 =====> Newman
5) 14001 =====> Bandingo
6) 14100 =====> Miniwiliger
```

```
-----
1) 12345 =====> Nowak
2) 13235 =====> Kowalski
3) 13444 =====> Schmidt
4) 14000 =====> Newman
5) 14001 =====> Bandingo
6) 14100 =====> Miniwiliger
```

Operatory projekcji - SelectMany

- Operator `SelectMany` spłaszczona strukturę składającą się z kolekcji elementów (która jest polem/właściwością wejściowej kolekcji)
 - Czyli implementuje interfejs `IEnumerable<T>`

```
public static void TestSelectMany()
{
    var resStud = Generator.GenerateStudentsWithTopicsEasy()
        .Where(s => s.Index < 20000)
        .SelectMany(s => s.Topics);
    resStud.ToList().ForEach(x => Console.Write(x + ";"));
    Console.WriteLine();
    Console.WriteLine(resStud.Count());
    var resChars = resStud
        .SelectMany(s => s);
    resChars.ToList().ForEach(x => Console.Write(x + ";"));
    Console.WriteLine();
    Console.WriteLine(resChars.Count());
}
```

```
public static void TestSelectManyQuery()
{
    // without filter is ok: from in after from in
    var resStud = from s in Generator.GenerateStudentsWithTopicsEasy()
        where s.Index < 20000
        from topic in s.Topics
        select topic;
    resStud.ToList().ForEach(x => Console.Write(x + ";"));
    Console.WriteLine();
    Console.WriteLine(resStud.Count());
    var resChars = from s in resStud
        from c in s
        select c;
    resChars.ToList().ForEach(x => Console.Write(x + ";"));
    Console.WriteLine();
    Console.WriteLine(resChars.Count());
}
```

C#;PHP;algorithms;C#;C++;fuzzy logic;Basic;Java;JavaScript;neural networks;Java;C#;algorithms;web programming;

14

C#;P;H;P;a;l;g;o;r;i;t;h;m;s;C#;C++;f;u;z;z;y; ;l;o;g;i;c;B;a;s;i;c;J;a;v;a;J;a;v;a;S;c;r;i;p;t;n;e;u;r;a;l; ;n;e;t;
w;o;r;k;s;J;a;v;a;C#;a;l;g;o;r;i;t;h;m;s;w;e;b; ;p;r;o;g;r;a;m;m;i;n;g;

96

SelectMany – z lambdą

- Zamiast rozbicia na składowe kolekcji , można wytworzyć elementy nowej kolekcji
- Wersją z drugą lambdą, która będzie wywołana dla każdego elementu „wewnętrznej” kolekcji

```
public static void TestSelectManyWith2Lambdas()
{
    var resStud = Generator.GenerateStudentsWithTopicsEasy()
        .Where(s => s.Index < 20000 && s.Name.Length <= 6)
        .SelectMany(s => s.Topics, (stud, topic) => new { stud.Name, topic });
    resStud.ToList().ForEach(Console.WriteLine);
    Console.WriteLine("-----");
    var resStud2 = from s in Generator.GenerateStudentsWithTopicsEasy()
        where s.Index < 20000 && s.Name.Length <= 6
        from topic in s.Topics
        select new { s.Name, topic };
    resStud2.ToList().ForEach(Console.WriteLine);
}
```

```
{ Name = Nowak, topic = C# }
{ Name = Nowak, topic = PHP }
{ Name = Nowak, topic = algorithms }
{ Name = Newman, topic = JavaScript }
{ Name = Newman, topic = neural networks }
-----
{ Name = Nowak, topic = C# }
{ Name = Nowak, topic = PHP }
{ Name = Nowak, topic = algorithms }
{ Name = Newman, topic = JavaScript }
{ Name = Newman, topic = neural networks }
```


Operatory porządkowania

- Istnieją 4 właściwe operatory porządkowania:
 - `OrderBy`
 - `OrderByDescending`
 - `ThenBy`
 - `ThenByDescending`
- Pierwsze dwa tworzą kolekcje uporządkowane implementujące interfejs `IOrderedEnumerable<T>`, których wymagają dwa kolejne, aby zapewnić stabilność wcześniejszego sortowania.
- Wersje ze słowem „Descending” sortują w porządku odwrotnym.
- Użycie sekwencji dowolnych dwóch pierwszych „anuluje” wcześniejsze sortowanie.
- W przypadku kwerendy LINQ
 - słowo kluczowe **descending** następuje po wyrażeniu sortującym (w tym samym miejscu można opcjonalnie użyć słowa **ascending**)
 - Kolejne składowe złożonego klucza oddzielone są przecinkiem
- Istnieje operator odwrócenia porządku: `Reverse`
- Wszystkie wymagają, w podstawowej wersji, wyrażenia lambda zwracającego klucz takiego typu `T`, że implementuje interfejs `IComparable<T>`.
 - Jednak to rozwiązanie umożliwia sortowanie tylko według jednej reguły porównywania np. dla typu `Student`

Operatory porządkowania - przykład

```
public static void TestOrderBy()
{
    var resStud = Generator.GenerateStudentsWithTopicsEasy()
        .OrderBy(s => s.Name)
        .ThenByDescending(s => s.Index);
    resStud.ToList().ForEach(Console.WriteLine);
    Console.WriteLine("-----");
    var resStud2 = from s in Generator.GenerateStudentsWithTopicsEasy()
        orderby s.Name, s.Index descending
        select s;
    resStud2.ToList().ForEach(Console.WriteLine);
}
```

```
5) 14001, Bandingo, Male, active, 3, topics: Java, C#,
2) 13235, Kowalski, Male, active, 1, topics: C#, C++, fuzzy logic,
6) 14100, Miniwiliger, Male, active, 2, topics: algorithms, web programming,
14) 24000, Newman, Female,no active, 1, topics: JavaScript, neural networks,
4) 14000, Newman, Female,no active, 3, topics: JavaScript, neural networks,
12) 23235, Nowak, Male, active, 1, topics: C#, C++, fuzzy logic,
11) 22345, Nowak, Female, active, 2, topics: C#, PHP, algorithms,
1) 12345, Nowak, Female, active, 1, topics: C#, PHP, algorithms,
13) 23444, Schmidt, Male,no active, 1, topics: Basic, Java,
3) 13444, Schmidt, Male,no active, 2, topics: Basic, Java,
-----
16) 24100, Bandingo, Male, active, 2, topics: algorithms, web programming,
15) 24001, Bandingo, Male, active, 3, topics: Java, C#,
5) 14001, Bandingo, Male, active, 3, topics: Java, C#,
2) 13235, Kowalski, Male, active, 1, topics: C#, C++, fuzzy logic,
6) 14100, Miniwiliger, Male, active, 2, topics: algorithms, web programming,
14) 24000, Newman, Female,no active, 1, topics: JavaScript, neural networks,
4) 14000, Newman, Female,no active, 3, topics: JavaScript, neural networks,
12) 23235, Nowak, Male, active, 1, topics: C#, C++, fuzzy logic,
11) 22345, Nowak, Female, active, 2, topics: C#, PHP, algorithms,
1) 12345, Nowak, Female, active, 1, topics: C#, PHP, algorithms,
13) 23444, Schmidt, Male,no active, 1, topics: Basic, Java,
3) 13444, Schmidt, Male,no active, 2, topics: Basic, Java,
```

Operatory porządkowania z komparatorem

- Gdy istnieje potrzeba posortowania wg innej metody niż domyślna dla danego typu, przydatna będzie wersja porządkowania z komparatorem jako drugi parametr.
- Nie istnieje wersja dla QES

```
class MyComparer : IComparer<string>
{
    int IComparer<string>.Compare(string x, string y)
    {
        return x.Length-y.Length;
    }
}

public static void TestOrderByWithComparer()
{
    var resStud = Generator.GenerateStudentsWithTopicsEasy()
        .OrderBy(s => s.Name, new MyComparer());
    resStud.ToList().ForEach(Console.WriteLine);
    //no version for Query expression
}
```

```
1) 12345,      Nowak, Female,   active, 1, topics: C#, PHP, algorithms,
11) 22345,      Nowak, Female,   active, 2, topics: C#, PHP, algorithms,
12) 23235,      Nowak,   Male,    active, 1, topics: C#, C++, fuzzy logic,
4) 14000,      Newman, Female,no active, 3, topics: JavaScript, neural networks,
14) 24000,      Newman, Female,no active, 1, topics: JavaScript, neural networks,
3) 13444,      Schmidt,  Male,no active, 2, topics: Basic, Java,
13) 23444,      Schmidt,  Male,no active, 1, topics: Basic, Java,
2) 13235,      Kowalski,  Male,   active, 1, topics: C#, C++, fuzzy logic,
5) 14001,      Bandingo,  Male,   active, 3, topics: Java, C#,
15) 24001,      Bandingo,  Male,   active, 3, topics: Java, C#,
16) 24100,      Bandingo,  Male,   active, 2, topics: algorithms, web programming,
6) 14100, Miniwiliger,  Male,   active, 2, topics: algorithms, web programming,
```

Operatory podziału

- Operatory pobierania lub odrzucania części kolekcji na podstawie pozycji:
 - Take
 - Skip
 - TakeWhile
 - SkipWhile
- Operatory „Take” pobierają elementy od początku kolekcji do danej pozycji (chyba, że kolekcja jest krótsza, wtedy tyle ile elementów istnieje) lub spełnienia warunku.
- Operatory „Skip” pomijają początkowe elementy do danej pozycji lub spełnienia warunku.
- Wywołując je w odpowiedniej kolejności można pobrać dowolny ciągły fragment kolekcji.
- Te operatory można by zrealizować operatorem `Where` z pozycją.
- Nie istnieje wersja QES

```
public static void TestTakeAndSkip()
{
    var resStud = Generator.GenerateStudentsWithTopicsEasy()
        .Skip(4).Take(5);
    resStud.ToList().ForEach(Console.WriteLine);
    //no version for Query expression
}
```

```
5) 14001,    Bandingo,    Male,    active, 3, topics: Java, C#,
6) 14100, Miniwiliger,    Male,    active, 2, topics: algorithms, web programming,
11) 22345,    Nowak, Female,    active, 2, topics: C#, PHP, algorithms,
12) 23235,    Nowak,    Male,    active, 1, topics: C#, C++, fuzzy logic,
13) 23444,    Schmidt,    Male, no active, 1, topics: Basic, Java,
```

Operatory SkipWhile, TakeWhile - przykłady

```
public static void TestTakeWhileAndSkipWhile()
{
    Generator.GenerateStudentsWithTopicsEasy().ToList().ForEach(Console.WriteLine);
    Console.WriteLine("-----");
    var resStud = Generator.GenerateStudentsWithTopicsEasy()
        .SkipWhile(s=>s.Active)
        .SkipWhile(s =>!s.Active)
        .TakeWhile(s=>s.Active);
    resStud.ToList().ForEach(Console.WriteLine);
    //no version for Query expression
}
```

```
1) 12345,      Nowak, Female, active, 1, topics: C#, PHP, algorithms,
2) 13235,      Kowalski, Male, active, 1, topics: C#, C++, fuzzy logic,
3) 13444,      Schmidt, Male,no active, 2, topics: Basic, Java,
4) 14000,      Newman, Female,no active, 3, topics: JavaScript, neural networks,
5) 14001,      Bandingo, Male, active, 3, topics: Java, C#,
6) 14100, Miniwiliger, Male, active, 2, topics: algorithms, web programming,
11) 22345,     Nowak, Female, active, 2, topics: C#, PHP, algorithms,
12) 23235,     Nowak, Male, active, 1, topics: C#, C++, fuzzy logic,
13) 23444,     Schmidt, Male,no active, 1, topics: Basic, Java,
14) 24000,     Newman, Female,no active, 1, topics: JavaScript, neural networks,
15) 24001,     Bandingo, Male, active, 3, topics: Java, C#,
16) 24100,     Bandingo, Male, active, 2, topics: algorithms, web programming,
-----
5) 14001,      Bandingo, Male, active, 3, topics: Java, C#,
6) 14100, Miniwiliger, Male, active, 2, topics: algorithms, web programming,
11) 22345,     Nowak, Female, active, 2, topics: C#, PHP, algorithms,
12) 23235,     Nowak, Male, active, 1, topics: C#, C++, fuzzy logic,
```

Wykonanie zapytania

- Część zapytań jest realizowana natychmiast
 - Np. Min, Max, ToList()
- Pozostałe są odłożone w czasie (ang. deferred/lazy evaluation):
 - Np. Select, Where, Take, Skip

```
public static void TestLazyExecution()
{
    var studs = Generator.GenerateStudentsWithTopicsEasy();
    var resStud = from s in studs
                   where s.Index < 20000 && s.Name.Length <= 6
                   select s;

    studs.Add(new StudentWithTopics(30, 15000, "Wuc", Gender.Male, true, 1,
                                     new List<string> { "C#", "Java", "algorithms" }));

    resStud.ToList().ForEach(Console.WriteLine);

    Console.WriteLine("-----");
    var resStud2 = (from s in studs
                    where s.Index < 20000 && s.Name.Length <= 6
                    select s).Count();

    studs.Add(new StudentWithTopics(31, 15001, "Wow", Gender.Female, true, 1,
                                     new List<string> { "C#" }));

    Console.WriteLine(resStud2);
}
```

```
1) 12345,      Nowak, Female,   active, 1, topics: C#, PHP, algorithms,
4) 14000,      Newman, Female, no active, 3, topics: JavaScript, neural networks,
30) 15000,     Wuc,   Male,    active, 1, topics: C#, Java, algorithms,
-----
3
```

Operatory grupowania

- Dla QES
- Podobne w działaniu do Lookup
 - Typ `IEnumerable<IGrouping<X,Y>>`

```
public static void TestGroupBy()
{
    var resStud = from s in Generator.GenerateStudentsWithTopicsEasy()
                  group s by s.DepartmentId;

    foreach (var dept in resStud)
    {
        Console.WriteLine(dept.Key);
        dept.ToList().ForEach(s => Console.WriteLine("  " + s));
    }
}
```

```
1
  1) 12345,      Nowak, Female,  active, 1, topics: C#, PHP, algorithms,
  2) 13235,      Kowalski,  Male,   active, 1, topics: C#, C++, fuzzy logic,
12) 23235,      Nowak,    Male,   active, 1, topics: C#, C++, fuzzy logic,
13) 23444,      Schmidt,  Male,no active, 1, topics: Basic, Java,
14) 24000,      Newman,  Female,no active, 1, topics: JavaScript, neural networks,
2
  3) 13444,      Schmidt,  Male,no active, 2, topics: Basic, Java,
  6) 14100, Miniwiliger,  Male,   active, 2, topics: algorithms, web programming,
11) 22345,      Nowak,    Female, active, 2, topics: C#, PHP, algorithms,
16) 24100,      Bandingo,  Male,   active, 2, topics: algorithms, web programming,
3
  4) 14000,      Newman,  Female,no active, 3, topics: JavaScript, neural networks,
  5) 14001,      Bandingo,  Male,   active, 3, topics: Java, C#,
15) 24001,      Bandingo,  Male,   active, 3, topics: Java, C#,
```

Dalsze użycie grup

- Podział na grupy może być wg złożonego klucza
- Aby skorzystać z grup w ramach QES LINQ należy użyć klauzuli **into** wraz z nazwą tego podziału na grupy.
- Załóżmy, że chcemy podzielić studentów w zależności od tego czy są aktywni i jaką mają płeć (czyli na 4 grupy), posortować wg tego klucza i dodatkowo w ramach każdej grupy posortować wg nazwiska.

```
public static void TestGroupByComplex()
{
    var resStud = from s in Generator.GenerateStudentsWithTopicsEasy()
                  group s by new { s.Active, s.Gender } into sGroup
                  orderby sGroup.Key.Active, sGroup.Key.Gender
                  select new
                  {
                      Active = sGroup.Key.Active,
                      sGroup.Key.Gender, // simpler
                      Students = sGroup.OrderBy(s=>s.Name)
                  };

    foreach (var group in resStud)
    {
        Console.WriteLine((group.Active ? "active" : "no active") + "      "+group.Gender);
        group.Students.ToList().ForEach(s => Console.WriteLine("    " + s));
    }
}
```


Zapytanie dla grup – wynik działania

```
no active      Female
  4) 14000,      Newman, Female,no active, 3, topics: JavaScript, neural networks,
 14) 24000,      Newman, Female,no active, 1, topics: JavaScript, neural networks,
no active      Male
  3) 13444,      Schmidt,  Male,no active, 2, topics: Basic, Java,
 13) 23444,      Schmidt,  Male,no active, 1, topics: Basic, Java,
active         Female
  1) 12345,      Nowak, Female,  active, 1, topics: C#, PHP, algorithms,
 11) 22345,      Nowak, Female,  active, 2, topics: C#, PHP, algorithms,
active         Male
  5) 14001,      Bandingo,  Male,  active, 3, topics: Java, C#,
 15) 24001,      Bandingo,  Male,  active, 3, topics: Java, C#,
 16) 24100,      Bandingo,  Male,  active, 2, topics: algorithms, web programming,
  2) 13235,      Kowalski,  Male,  active, 1, topics: C#, C++, fuzzy logic,
  6) 14100, Miniwiliger, Male,  active, 2, topics: algorithms, web programming,
 12) 23235,      Nowak,    Male,  active, 1, topics: C#, C++, fuzzy logic,
```

Operatory elementów

- Mamy kilka operatorów jednego elementu z kolekcji:
 - `First / FirstOrDefault`
 - `Last / LastOrDefault`
 - `ElementAt / ElementAtOrDefault`
 - `Single / SingleOrDefault`
 - `DefaultIfEmpty`
- Wersje bez „Default” w razie nieobecności danego elementu zgłaszają wyjątek
- Wersje z „Default” zwracają wartość domyślną danego typu.
- „Single” oznacza, że jest dokładnie jeden element w kolekcji
- `DefaultIfEmpty` dla niepustej kolekcji zwraca tę kolekcję, dla pustej kolekcję jednoelementową z domyślną wartością.

Łączenie różnych kolekcji - GroupJoin

- Jak łączenie z kluczem obcym
- Połączmy wydziały i studentów
 - Dokładnie: do wydziałów przydzielamy grupy studentów
- Operator `GroupJoin()`
- Klauzula **`join ... on... equals... into ...`**
- Należy podać, które klucze połączyć
 - W klauzuli zamiast operatora porównania `'=='` używa się słowa kluczowego **`equals`**

```
public static void TestGroupJoin()
{
    var resStud = Generator.GenerateDepartmentsEasy()
        .GroupJoin(Generator.GenerateStudentsWithTopicsEasy(),
            dept => dept.Id,
            stud => stud.DepartmentId,
            (department, students) => new
            {
                Department = department,
                Students = students
            }
        );

    foreach (var group in resStud)
    {
        Console.WriteLine(group.Department.Name);
        group.Students.ToList().ForEach(s => Console.WriteLine("  " + s));
    }
}
```

Łączenie różnych kolekcji - join ... on... equals... into ...

```
Console.WriteLine("-----");  
var resStud2 = from d in Generator.GenerateDepartmentsEasy()  
               join s in Generator.GenerateStudentsWithTopicsEasy()  
               on d.Id equals s.DepartmentId into dGroup  
               select new  
               {  
                   Department = d,  
                   Students = dGroup  
               };  
foreach (var group in resStud2)  
{  
    Console.WriteLine(group.Department.Name);  
    group.Students.ToList().ForEach(s => Console.WriteLine("  " + s));  
}
```

```
Computer Science  
  1) 12345,      Nowak, Female,  active, 1, topics: C#, PHP, algorithms,  
  2) 13235,      Kowalski,  Male,   active, 1, topics: C#, C++, fuzzy logic,  
12) 23235,      Nowak,    Male,   active, 1, topics: C#, C++, fuzzy logic,  
13) 23444,      Schmidt,  Male,no active, 1, topics: Basic, Java,  
14) 24000,      Newman, Female,no active, 1, topics: JavaScript, neural networks,  
Electronics  
  3) 13444,      Schmidt,  Male,no active, 2, topics: Basic, Java,  
  6) 14100, Miniwiliger,  Male,   active, 2, topics: algorithms, web programming,  
11) 22345,      Nowak,    Female, active, 2, topics: C#, PHP, algorithms,  
16) 24100,      Bandingo,  Male,   active, 2, topics: algorithms, web programming,  
Mathematics  
  4) 14000,      Newman, Female,no active, 3, topics: JavaScript, neural networks,  
  5) 14001,      Bandingo,  Male,   active, 3, topics: Java, C#,  
15) 24001,      Bandingo,  Male,   active, 3, topics: Java, C#,  
Mechanics  
-----  
Computer Science  
  1) 12345,      Nowak, Female,  active, 1, topics: C#, PHP, algorithms,  
  2) 13235,      Kowalski,  Male,   active, 1, topics: C#, C++, fuzzy logic,
```

Operator Join dla tworzenie płaskiej kolekcji

- Łączenie dwóch kolekcji na podstawie tego samego klucza w płaską strukturę
- Operator `Join()`, w klauzuli brak słowa kluczowego dla grupy: **into...**,

```
public static void TestJoin()
{
    var studs = Generator.GenerateStudentsWithTopicsEasy();
    // there are no 6 department
    studs.Add(new StudentWithTopics(30, 15000, "Wuc", Gender.Male, true, 6,
        new List<string> { "C#", "Java", "algorithms" }));
    var resStud = studs
        .Join(Generator.GenerateDepartmentsEasy(),
            stud => stud.DepartmentId,
            dept => dept.Id,
            (student, department) => new
            {
                DepartmentName = department.Name,
                StudentName = student.Name
            })
        );
    foreach (var elem in resStud){
        Console.WriteLine($"{elem.DepartmentName} -> {elem.StudentName}");
        Console.WriteLine("-----");
    }
    var resStud2 = from s in Generator.GenerateStudentsWithTopicsEasy()
        join d in Generator.GenerateDepartmentsEasy()
        on s.DepartmentId equals d.Id
        select new
        {
            DepartmentName = d.Name,
            StudentName = s.Name
        };
    foreach (var elem in resStud2){
        Console.WriteLine($"{elem.DepartmentName} -> {elem.StudentName}");
    }
```

```
Computer Science -> Nowak
Computer Science -> Kowalski
Electronics -> Schmidt
Mathematics -> Newman
Mathematics -> Bandingo
Electronics -> Miniwiliger
Electronics -> Nowak
Computer Science -> Nowak
Computer Science -> Schmidt
Computer Science -> Newman
Mathematics -> Bandingo
Electronics -> Bandingo
-----
Computer Science -> Nowak
Computer Science -> Kowalski
Electronics -> Schmidt
Mathematics -> Newman
Mathematics -> Bandingo
Electronics -> Miniwiliger
Electronics -> Nowak
Computer Science -> Nowak
Computer Science -> Schmidt
Computer Science -> Newman
Mathematics -> Bandingo
Electronics -> Bandingo
```

Operator SelectMany, iloczyn kartezjański

- Kolekcja wszystkich par (x,y), gdzie x z pierwszego zbioru, a y z drugiego zbioru.
- Istnieje wersja wykorzystaniem indeksu elementu
- To samo można uzyskać z operatorem Join, gdzie zamiast kluczy zwracane będzie zawsze **true**.

```
public static void CartesianProduct()
{
    var resCart = from num in Generator.GenerateIntsEasy()
                  where num % 2 == 0
                  from d in Generator.GenerateNamesEasy()
                  where d.Length < 7
                  select new
                  {
                      Number = num,
                      Word=d
                  };
    foreach (var elem in resCart)
    {
        Console.WriteLine($"{elem.Number} -> {elem.Word}");
    }
    Console.WriteLine("-----");
    var resCart2 = Generator.GenerateIntsEasy()
        .Where(num => num % 2 == 0)
        .SelectMany(
            s=>Generator.GenerateNamesEasy().Where(s=>s.Length < 7),
            (n,s)=>new {
                Number = n,
                Word = s
            });
    foreach (var elem in resCart2)
    {
        Console.WriteLine($"{elem.Number} -> {elem.Word}");
    }
}
```

```
2 -> Nowak
2 -> Newman
6 -> Nowak
6 -> Newman
8 -> Nowak
8 -> Newman
-----
{ Number = 2, Word = Nowak }
{ Number = 2, Word = Newman }
{ Number = 6, Word = Nowak }
{ Number = 6, Word = Newman }
{ Number = 8, Word = Nowak }
{ Number = 8, Word = Newman }
```

Operatory zbiorowe

- Kilka operatorów zbiorowych:
 - `Distinct`
 - Usuwa powtarzające się elementy
 - `Union`
 - Suma dwóch kolekcji tych samych typów
 - `Intersect`
 - Część wspólna dwóch kolekcji tych samych typów
 - `Except`
 - Pierwsza kolekcja minus druga kolekcja tych samych typów
- Wszystkie operacje zakładają zaimplementowanie metody **`bool Equals(object other)`** dla typu **`T`**
 - Domyślna implementacja tej metody w klasie `Object` zakłada identyczność referencji
- Gdy metoda porównująca nie została zaimplementowana w klasie, lub chcemy użyć innego porównywania, wszystkie te operatory zbiorowe LINQ posiadają wersję z parametrem, obiektem klasy implementującej interfejs `IEqualityComparer<T>`, który posiada metody:
 - **`public bool Equals<T>(T x, T y)`**
 - **`public int GetHashCode<T>(T x)`**
- Dla klas anonimowych wytworzony zostanie metoda porównująca pole po polu, czyli najczęściej to czego oczekujemy.
 - Zamiast pisać klasę porównującą z metodami, można przekształcić dane na klasy anonimowe
- Nie istnieje wersja dla klauzul LINQ.

Operator Distinct()

- Założenie: metoda Equals() **nie została** zaimplementowana w klasie Student.

```
class Comp : IEqualityComparer<Student>
{
    public bool Equals(Student x, Student y)
    {
        return x.Id == y.Id;
    }
    public int GetHashCode(Student obj)
    {
        return obj.Id.GetHashCode();
    }
}
```

1) 12345,	Nowak, Female,	active, 1, topics: C#, PHP, algorithms,
2) 13235,	Kowalski, Male,	active, 1, topics: C#, C++, fuzzy logic,
1) 12345,	Nowak, Female,	active, 1, topics: C#, PHP, algorithms,

1) 12345,	Nowak, Female,	active, 1, topics: C#, PHP, algorithms,
2) 13235,	Kowalski, Male,	active, 1, topics: C#, C++, fuzzy logic,

```
public static void TestDistinct()
{
    var set1 = Generator.GenerateStudentsWithTopicsEasy()
        .Where(s => s.Id >= 0 && s.Id <=2)
        .ToList();
    set1.Add(new Student(1, 12345, "Nowak", Gender.Female, true, 1,
        new List<string> { "C#", "PHP", "algorithms" })); // copy od first student

    set1.Distinct().ToList().ForEach(Console.WriteLine);
    Console.WriteLine("-----");
    set1.Distinct(new Comp()).ToList().ForEach(Console.WriteLine);
}
```


Operator Union

- Typy **anonimowe** mają zaimplementowane metody do **porównywania** obiektów.

1)	12345,	Nowak,	Female,	active
2)	13235,	Kowalski,	Male,	active
3)	13444,	Schmidt,	Male,	no active
4)	14000,	Newman,	Female,	no active
3)	13444,	Schmidt,	Male,	no active
4)	14000,	Newman,	Female,	no active
5)	14001,	Bandingo,	Male,	active
6)	14100,	Miniwiliger,	Male,	active

1)	12345,	Nowak,	Female,	active
2)	13235,	Kowalski,	Male,	active
3)	13444,	Schmidt,	Male,	no active
4)	14000,	Newman,	Female,	no active
5)	14001,	Bandingo,	Male,	active
6)	14100,	Miniwiliger,	Male,	active

```
public static void TestUnion()
{
    var set1 = Generator.GenerateStudentsEasy()
        .Where(s => s.Id >= 1 && s.Id <= 4);
    var set2 = Generator.GenerateStudentsEasy()
        .Where(s => s.Id >= 3 && s.Id <= 6);
    set1.Union(set2).ToList().ForEach(Console.WriteLine);
    Console.WriteLine("-----");
    set1.Union(set2, new Comp()).ToList().ForEach(Console.WriteLine);
}
```

```
public static void TestUnionAnonymous()
{
    var set1 = Generator.GenerateStudentsEasy()
        .Where(s => s.Id >= 1 && s.Id <= 4)
        .Select(s => new
        {
            s.Id, s.Index, s.Name
        });
    var set2 = Generator.GenerateStudentsEasy()
        .Where(s => s.Id >= 3 && s.Id <= 6)
        .Select(s => new
        {
            s.Id, s.Index, s.Name
        });
    set1.Union(set2).ToList().ForEach(Console.WriteLine);
}
```

```
{ Id = 1, Index = 12345, Name = Nowak }
{ Id = 2, Index = 13235, Name = Kowalski }
{ Id = 3, Index = 13444, Name = Schmidt }
{ Id = 4, Index = 14000, Name = Newman }
{ Id = 5, Index = 14001, Name = Bandingo }
{ Id = 6, Index = 14100, Name = Miniwiliger }
```

Operatory generacji, dołączania

- Istnieją operatory generacji kolekcji:
 - `IEnumerable.Range(int nFrom, int nTo)`
 - Tylko dla **int** kolekcja liczb od `nFrom` do `nTo-1`
 - `Repeat<T>(T elem, int n)`
 - Kolekcja `n` elementów `elem`.
 - `Empty<T>()`
 - Pusta kolekcja danego typu `T`
- Istnieje operacja dołączenia:
 - `Concat()`
 - Druga kolekcja dołączana jest po ostatnim elemencie pierwszej kolekcji
- Istnieje operacja porównywania sekwencji
 - `SequenceEqual()`
 - Sekwencje są równe jeśli długości mają równe i w obydwóch elementy są w tej samej kolejności
 - Można podać komparator jak w operatorach zbiorowych

Kwantyfikatory

- Istnieją kwantyfikatory zwracające wartość logiczną:
 - `All`
 - Z predykatem, sprawdza, czy wszystkie elementy spełniają predykat
 - `Any`
 - Bezparametryczny zwraca **false** tylko gdy kolekcja jest pusta
 - Z predykatem, sprawdza, czy jakikolwiek element spełnia predykat
 - `Contains`
 - Z jednym parametrem sprawdza, czy istnieje taki element używając metody `Equals()` z danego typu.
 - Drugim parametrem może być obiekt implementujący `IEqualityComparer<T>`

Klauzula **let** 1/2

- Klauzula **let** w QES pozwala na jednokrotne tworzenie zmiennych, które w innym przypadku trzeba by tworzyć wielokrotnie w ramach jednego zapytania.
- Można tworzyć wiele klauzul **let**, nawet jedna za drugą.
- Klauzule **let** muszą być po pierwszej klauzuli **from... in...** i przed ostatnio klauzulą **select** lub **group... by...**

```
static void TestClauseWithoutLet() {  
    ClauseWithoutLet(".", "*");  
}  
  
static void ClauseWithoutLet(string rootDirectory, string searchPattern)  
{  
    IEnumerable<string> filenames = Directory.GetFiles(rootDirectory, searchPattern);  
    var fileResults =  
        from fileName in filenames  
        orderby new FileInfo(fileName).Length, fileName  
        select new FileInfo(fileName);  
    foreach (var fileResult in fileResults)  
    {  
        Console.WriteLine($"{fileResult.Name} ({fileResult.Length})");  
    }  
}
```

```
LinqExamples.runtimeconfig.json (154)  
LinqExamples.runtimeconfig.dev.json (236)  
LinqExamples.deps.json (428)  
LinqExamples.pdb (18296)  
LinqExamples.dll (40960)  
LinqExamples.exe (174592)
```

Klauzula **let** 2/2

- Ten sam efekt bez tworzenia dwa razy obiektów typu FileInfo
- Kod QES z klauzulą **let**
- Kod MBS z dodatkową projekcją **Select** i zmianą projekcji w klauzuli **select** z QES

```
static void ClauseWithLet(string rootDirectory, string searchPattern)
{
    IEnumerable<string> filenames = Directory.GetFiles(rootDirectory, searchPattern);
    var fileResults =
        from fileName in filenames
        let file= new FileInfo(fileName)
        orderby file.Length, fileName
        select file;
    foreach (var fileResult in fileResults)
    {
        Console.WriteLine($"{fileResult.Name} ({fileResult.Length})");
    }
}
```

```
static void FluentApiWithLet(string rootDirectory, string searchPattern)
{
    IEnumerable<string> filenames = Directory.GetFiles(rootDirectory, searchPattern);
    var fileResults = filenames.Select(fileName => new { fileName, file = new FileInfo(fileName) })
        .OrderBy(elem => (elem.file.Length, elem.fileName))
        .Select(elem => elem.file);
    foreach (var fileResult in fileResults)
    {
        Console.WriteLine($"{fileResult.Name} ({fileResult.Length})");
    }
}
```

```
LinqExamples.runtimeconfig.json (154)
LinqExamples.runtimeconfig.dev.json (236)
LinqExamples.deps.json (428)
LinqExamples.pdb (18296)
LinqExamples.dll (40960)
LinqExamples.exe (174592)
```

Przykład zapytania do bazy w LINQPad

- Narzędzie nie podpowiada nazw pól, metod itd. (brak IntelliSense), ale je sprawdza
- Potrafi przekształcać zapis QES/MBS na:
 - MBS
 - Zapytanie SQL, które trafi do bazy
 - Na język pośrednie IL
 - Na drzewo zapytania
 - Można zaznaczyć fragment zapytania do pokazania fragmentu drzewa
- Można też używać dla kolekcji nie opartych na bazie danych

LINQPad 6

File Edit Query Debug Help

Customers

Employees

- EmployeeID (Int32)
- LastName (String)
- FirstName (String)
- Title (String)
- BirthDate (DateTime?)
- HireDate (DateTime?)
- Address (String)
- City (String)
- Region (String)
- PostalCode (String)
- Country (String)
- HomePhone (String)
- Extension (String)
- Photo (Binary)
- Notes (String)
- ReportsTo (Int32?)

Orders

OrderDetails

My Queries Samples

Query 1* (Employees where)* +

Language C# Expression

```

from e in Employees
where e.FirstName.Length > 6
select new {
    e.FirstName,
    e.LastName,
    e.HomePhone
}

```

Results SQL IL Tree

First Name	Last Name	Home Phone
Margaret	Peacock	(206) 555-8122
Michael	Suyama	(71) 555-7773
Caroline	Patterson	(206) 555-3487
Laurent	Pereira	88 01 01 68

Results SQL IL Tree

```

Employees
.Where (e => (e.FirstName.Length > 6))
.Select (
    e =>
        new
        {
            FirstName = e.FirstName,
            LastName = e.LastName,
            HomePhone = e.HomePhone
        }
)

```

Results SQL IL Tree

```

-- Region Parameters
-- @p0: Int32 [6]
-- EndRegion
SELECT [t0].[First Name] AS [FirstName], [t0].[Last Name] AS [LastName], [t0].[Home Phone] AS [HomePhone]
FROM [Employees] AS [t0]
WHERE LEN([t0].[First Name]) > @p0

```

Results SQL IL Tree

Refresh Expand All Collapse All SyntaxNode SyntaxToken SyntaxTrivia Dump Diagnostics

Kind	Property Name	Span
CompilationUnit		(0,107)
GlobalStatement		(0,107)
ExpressionStatement	Statement	(0,107)
QueryExpression	Expression	(0,107)
FromClause	FromClause	(0,20)
FromKeyword	FromKeyword	(0,5)
Trail: WhitespaceTrivia		(4,5)
IdentifierToken	Identifier	(5,7)
Trail: WhitespaceTrivia		(6,7)
InKeyword	InKeyword	(7,10)
Trail: WhitespaceTrivia		(9,10)
IdentifierToken	Identifier	(10,20)
IdentifierToken	Identifier	(10,20)
Trail: EndOfLineTrivia		(19,20)
QueryBody	Body	(20,107)
WhereClause	WhereClause	(20,48)
WhereKeyword	WhereKeyword	(20,26)
Trail: WhitespaceTrivia		(25,26)
GreaterThanExpression	Condition	(26,48)
SimpleMemberAccessExpression	Left	(26,44)
SimpleMemberAccessExpression	Expression	(26,37)
IdentifierToken	Expression	(26,27)
IdentifierToken	Identifier	(26,27)

CompilationUnitSyntax

C#

MECHANIZM REFLEKSJI

Mechanizm refleksji

- Zdecydowana większość klas z tej tematyki znajduje się w module `System.Reflection`
- Pobranie typu elementu za pomocą:
 - Metody `GetType()`, która jest obecna od klasy `Object`
 - Można używać tylko dla referencji do obiektu
 - Operatora `typeof(arg)`
 - Gdy jest to klasa statyczna, typ **enum** itp.
 - Metody statycznej `GetType(string typeName)` – podając kwalifikowaną nazwę typu
- Zwraca jest wartość typu `Type`.

Obiekty Type

- Dla każdej klasy tworzony jest **jeden** obiekt typu `System.Type`, stąd można porównywać typy dwóch obiektów za pomocą operatora `==`.
- Obiekt ten pozwala:
 - Sprawdzić nazwę typu (właściwość `Type.Name`)
 - Czy typ jest publiczny (właściwość `Type.IsPublic`), czy jest abstrakcyjny, tablicą, typem referencyjnym, typem enum itd.
 - Jaki jest typ bazowy (właściwość `Type.BaseType`)
 - Jak interfejsy obsługuje (metoda `Type.GetInterfaces()`)
 - Podzespół, gdzie był zdefiniowany (właściwość `Type.Assembly`), przestrzeń nazw itp..
 - Jakie są właściwości, metody, pola i inne składowe typu (metody `Type.GetProperties()`, `Type.GetMethods()`, `Type.GetFields()` i inne)
 - Znaleźć właściwość, metodę, pole o podanej nazwie
 - Jakimi atrybutami opatrzony jest typ (metoda `Type.GetCustomAttributes()`)
 - Elementy związane z typami generycznymi
- Obiekt ten **nie zawiera** metod rozszerzających – są one składowymi statycznym typu, gdzie zostały zaimplementowane.

Mechanizm refleksji - przykład

```
public static void DataTypeReflection()
{
    DateTime dateTime = new DateTime();
    Type type = dateTime.GetType();
    foreach (System.Reflection.PropertyInfo property in type.GetProperties())
    {
        Console.WriteLine(property.Name);
    }
}
```

Date
Day
DayOfWeek
DayOfYear
Hour
Kind
Millisecond
Minute
Month
Now
Second
Ticks
TimeOfDay
Today
Year
UtcNow

Type.Name

- Przykład nazw

```
public static void NameDemo()  
{  
    object[] values = { "str", false, 100, 3.14, 'z' };  
    foreach (var value in values)  
        Console.WriteLine("{0} - has type's name {1}, FullName={2}", value,  
            value.GetType().Name, value.GetType().FullName);  
}
```

```
str - has type's name String, FullName=System.String  
False - has type's name Boolean, FullName=System.Boolean  
100 - has type's name Int32, FullName=System.Int32  
3,14 - has type's name Double, FullName=System.Double  
z - has type's name Char, FullName=System.Char
```

Refleksja – przykład z własną klasą

```
public static void BaseInfo()
{
    Person person = new Person();
    Type info = person.GetType();
    Console.WriteLine($"Name: {info.Name}");
    Console.WriteLine($"IsPublic: {info.IsPublic}");
    Console.WriteLine($"BaseType : {info.BaseType}");
    var props=info.GetProperties();
    Console.WriteLine($"Properties :");

    foreach (var prop in props)
    {
        Console.WriteLine($"  Name : {prop.Name}");
        Console.WriteLine($"  Name : {prop.PropertyType.Name}");
    }

    Console.WriteLine($"Methods :");
    var meths= info.GetMethods();

    foreach (var meth in meths)
    {
        Console.WriteLine($"  Name : {meth.Name}");
        Console.WriteLine($"  Params amount : {meth.GetParameters().Length}"); // number of parameter
    }
}
```

```
class Person
{
    [DisplayName("Wynagrodzenie")]
    public int Salary { get; set; }

    [Obsolete]
    public string Show(string comment, int starsNo)
    {
        string addStr = "";
        for (int i = 0; i < starsNo; i++)
            addStr += '*';
        return comment + addStr + Salary + addStr;
    }
}
```

Refleksja - Wynik

```
Name: Person
IsPublic: False
BaseType : System.Object
Properties :
  Name : Salary
  Name : Int32
Methods :
  Name : get_Salary
  Params amount : 0
  Name : set_Salary
  Params amount : 1
  Name : Show
  Params amount : 2
  Name : GetType
  Params amount : 0
  Name : ToString
  Params amount : 0
  Name : Equals
  Params amount : 1
  Name : GetHashCode
  Params amount : 0
```

- Są wszystkie metody, w tym setter i getter
- W metodzie Show widać, że są 2 parametry
- Można zobaczyć co to za parametry: nazwy, typy itd., gdyż zwracana jest tablica typu `Type`.

Metody `GetMethod` i `Invoke`

- Można pobrać obiekt związany z metodą przeglądanego obiektu np. za pomocą metody `Type.GetMethod(...)`.
 - Istnieje wiele metod o tej nazwie lub podobnej
- Wracana wartość jest typu `MethodInfo`. Zawiera metody do sprawdzenia, czy metoda jest publiczna, statyczna itd.
- Pozwala sprawdzić parametry metody, ich typy, sposób przekazywania do metody (**ref**, **out** itp.). Podobnie można sprawdzić cechy zwracanego wyniku.
- Posiadając obiekt typu `MethodInfo` można wywołać tą metodę dla konkretnego obiektu i podanych argumentów

GetMethod i Invoke - przykład

- Demonstracja pobrania i wywołania metody `Substring()` dla obiektu `str` o parametrach (4,4)

```
public static void InvokeDemo()
{
    string str = "big demonstration";
    MethodInfo methodInfo=str.GetType().GetMethod("Substring",
        new Type[] { typeof(int), typeof(int) });
    // metoda "Substring" z dwoma parametrami typu "int"
    string result = (string)methodInfo.Invoke(str, new object[] { 4, 4 });
    Console.WriteLine($"Result = {result}");
}
```

Result = demo

Operator `nameof`

- Często podczas różnych operacji chcemy użyć nazwy klasy, metody, właściwości itd. jako ciągu znaków (**string**). Np. chcemy w logu zapisać, że rozpoczynamy działanie metody `SpecialMethod`. Zamiast kopiować tekstowo nazwę metody do loga i tworzyć kod poniższy:

```
Logger.write(log, "Begin of SpecialMethod");
```

- lepiej użyć **`nameof`**, która przekształci argument będący metodą (itp.) w jej nazwę:

```
Logger.write(log, "Begin of " + nameof(SpecialMethod));
```

- Zalety:
 - IntelliSense podpowie dostępne metody (nie trzeba robić kopiuj-wklej)
 - Nie zrobimy literówki (bo nie przejdzie kompilacja)
 - W przypadku refaktoryzacji kodu i np. zmiany nazwy metody za pomocą inteligentnej zmiany w całym projekcie nie trzeba będzie szukać i zmieniać tej linijki kodu ręcznie.
 - Szukając referencji do tej metody znajdziemy też tą linijkę.

Klasy `Assembly` i inne

- Można również przeglądać dane nt. podzespołu poprzez klasę `Assembly` i jej metody.
- Klasa `Assembly` pozwala na tworzenie obiektów określonego typu poprzez metodę `CreateInstance`.
- Klasa ta pozwala też na dostęp do atrybutów.
- Posiada metody pozwalające dowiedzieć się o klasie (zestawie) informacji niemożliwych z obiektu klasy `Type`.
- M. in. nie wymaga instancji obiektu, aby działać.
 - Np. Pozwala sprawdzić, czy obiekt był skompilowany dla wersji 32-bitowej czy 64-bitowej.