



# EntityFramework LINQ2Entities

Leszek Siwik

- *Entity Framework is Microsoft's recommended data access technology for new applications*
- Entity Framework (EF) is an object-relational mapper that enables .NET developers to work with relational data using domain-specific objects. It eliminates the need for most of the data-access code that developers usually need to write.

- Są trzy główne podejścia w tworzeniu aplikacji bazodanowych z wykorzystaniem EF:
  - **Code first**
    - polega na napisaniu klas POCO, a następnie na ich podstawie wygenerowaniu bazy danych
  - **Model first**
    - polega na zaprojektowaniu modelu w EDMX (Entity Data Model XML) lub w designerze, a następnie na jego podstawie wygenerowaniu DDL tworzącego tabele i powiązania w bazie
  - **Database first**
    - Polega na wygenerowaniu EDM na podstawie struktury bazy danych
- <https://dotsub.com/view/9e7528b5-e1a7-4cdc-996f-b2928a822375>
- <https://msdn.microsoft.com/en-us/data/jj590134>

# CodeFirst vs ModelFirst vs DatabaseFirst



- <https://msdn.microsoft.com/en-us/data/jj193542>
- <https://msdn.microsoft.com/en-us/data/jj205424>
- <https://msdn.microsoft.com/en-us/data/jj206878>

- Tworzymy klasę z property które mają być mapowane na atrybuty w bazie danych. Np. jak niżej:

```
class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
}
```

- Tworzymy klasę kontekstową dziedziczącą po DbContext zawierającą zbiory (DbSet) kolekcji które mają być zarządzane przez EF:

```
class BlogContext:DbContext
{
    public DbSet<Post> Posts { set; get; }
}
```

# Zapytania o blogi



```
// pobierz wszystkie blogi z bazy i posortuj malejaco po
nazwie
IQueryable<string> query = from b in db.Blogs
                           orderby b.Name descending
                           select b.Name;

//wypisz wszystkie pobrane blogi
foreach (var item in query)
{
    Console.WriteLine(item);
}

Console.WriteLine("Blogi z method synthax");

IQueryable<string> blogNames = db.Blogs.Select(b=>b.Name);
//wypisz wszystkie pobrane blogi z lambda
foreach (String bl in blogNames)
{
    Console.WriteLine(bl);
}
```

(localdb)\mssqllocaldb (SQL Server 13.0.4)

- Databases
  - System Databases
  - Database Snapshots
  - BlogEF.BlogContext
    - Database Diagrams
    - Tables
      - System Tables
      - FileTables
      - dbo.\_\_MigrationHistory
      - dbo.Blogs
        - Columns
          - BlogID (PK, int, not null)
          - Name (nvarchar(max), null)
        - Keys
          - PK\_dbo.Blogs
        - Constraints
        - Triggers
        - Indexes
          - PK\_dbo.Blogs (Clustered)

SQL:BatchStarting	create database [BlogEF.BlogContext]
SQL:BatchCompleted	create database [BlogEF.BlogContext]
Audit Logout	
RPC:Completed	exec sp_reset_connection
Audit Login	-- network protocol: Named Pipes set quote...
SQL:BatchStarting	if serverproperty('EngineEdition') <> 5 exe...
SQL:BatchCompleted	if serverproperty('EngineEdition') <> 5 exe...
Audit Login	-- network protocol: Named Pipes set quote...
SQL:BatchStarting	CREATE TABLE [dbo].[Blogs] ( [BlogID] ...

```

create database [BlogEF.BlogContext]
go
CREATE TABLE [dbo].[Blogs] (
    [BlogID] [int] NOT NULL IDENTITY,
    [Name] [nvarchar](max),
    CONSTRAINT [PK_dbo.Blogs] PRIMARY KEY ([BlogID])
)
go

```

# Adnotacje

**Za:**

<http://www.entityframeworktutorial.net/code-first/dataannotation-in-code-first.aspx>

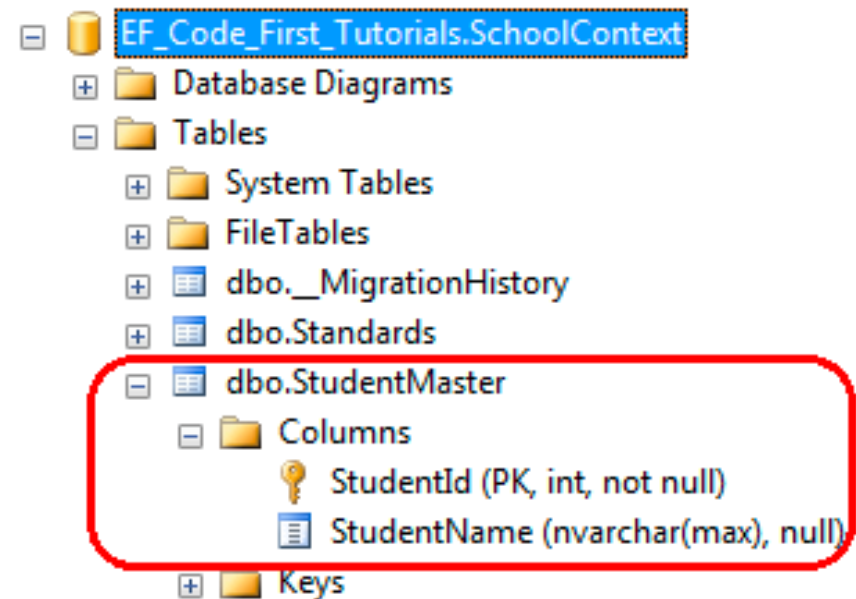
<https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/data-annotations>



**[Table(string name, Properties:[Schema = string])]**

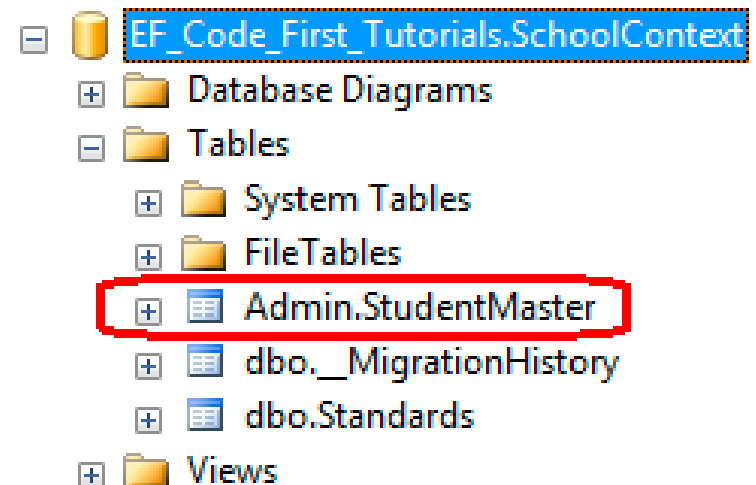
```
using System.ComponentModel.DataAnnotations.Schema;

[Table("StudentMaster")]
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
}
```



**[Table(string name, Properties:[Schema = string])]**

```
[Table("StudentMaster", Schema="Admin")]  
public class Student  
{  
    public int StudentID { get; set; }  
    public string StudentName { get; set; }  
}
```

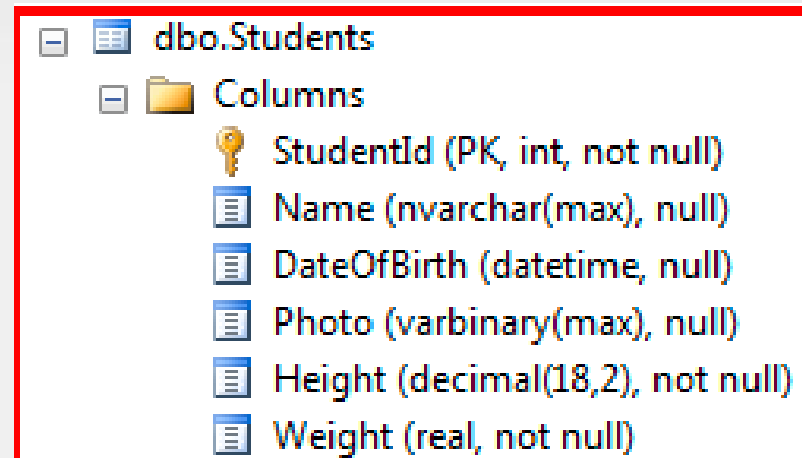


# DataAnnotations - Column

**[Column (string name, Properties:[Order = int],[TypeName = string])]**

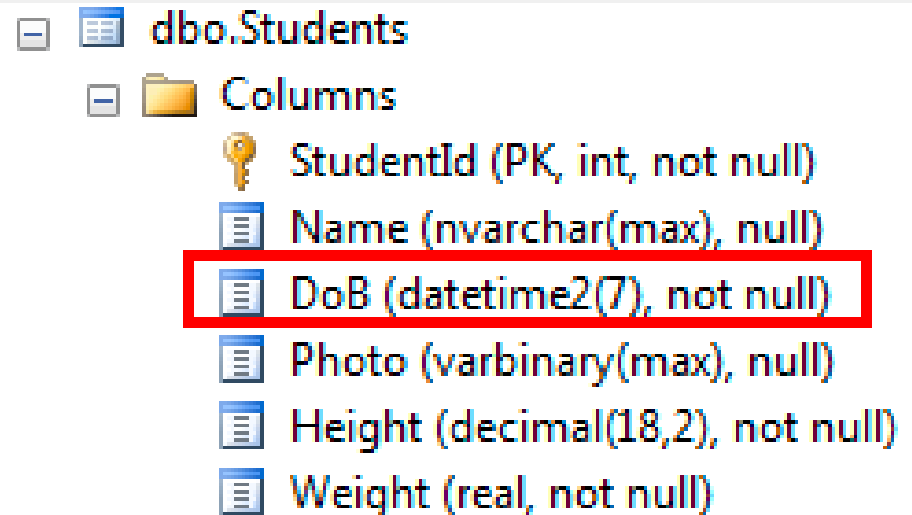
```
public class Student
{
    public int StudentID { get; set; }

    [Column("Name")]
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[] Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }
}
```



```
public class Student
{
    public int StudentID { get; set; }

    [Column("Name")]
    public string StudentName { get; set; }
    [Column("DoB", TypeName="DateTime2")]
    public DateTime DateOfBirth { get; set; }
    public byte[] Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }
}
```



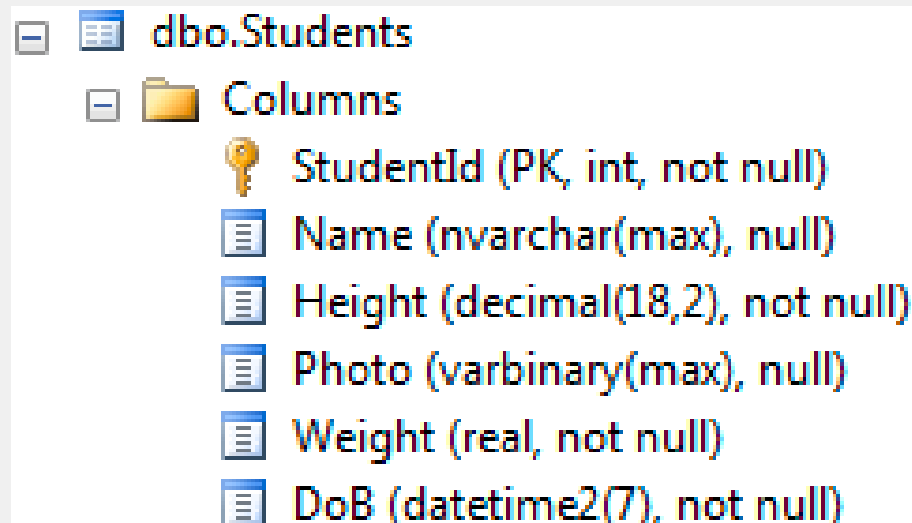
# DataAnnotations - Column

**[Column (string name, Properties:[Order = int],[TypeName = string])]**

```
public class Student
{
    [Column(Order = 0)]
    public int StudentID { get; set; }

    [Column("Name", Order = 1)]
    public string StudentName { get; set; }

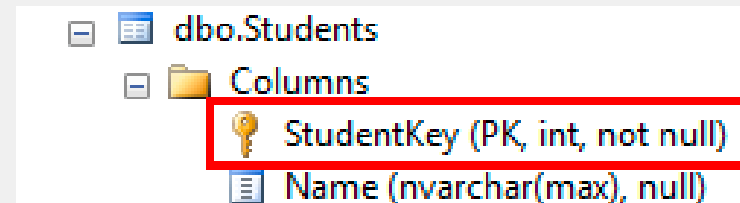
    [Column("DoB", Order = 5)]
    public DateTime DateOfBirth { get; set; }
    [Column(Order = 3)]
    public byte[] Photo { get; set; }
    [Column(Order = 2)]
    public decimal Height { get; set; }
    [Column(Order = 4)]
    public float Weight { get; set; }
}
```



W EF przyjęto konwencję zgodnie z którą jako klucz główny ustawiany jest w bazie atrybut nazwany w klasie jako **id** lub **<Entity Class Name>Id**

Adnotacją **Key** można zmienić to domyślne zachowanie

```
public class Student
{
    [Key]
    public int StudentKey { get; set; }
    public string StudentName { get; set; }
}
```

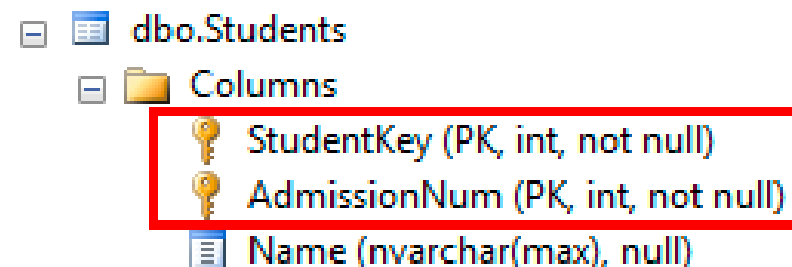


Od EFv6 poprzez kombinacje adnotacji **Key** oraz **Column(Order)** możliwe jest tworzenie **kluczy złożonych**

```
public class Student
{
    [Key]
    [Column(Order=1)]
    public int StudentKey { get; set; }

    [Key]
    [Column(Order=2)]
    public int AdmissionNum { get; set; }

    public string StudentName { get; set; }
}
```



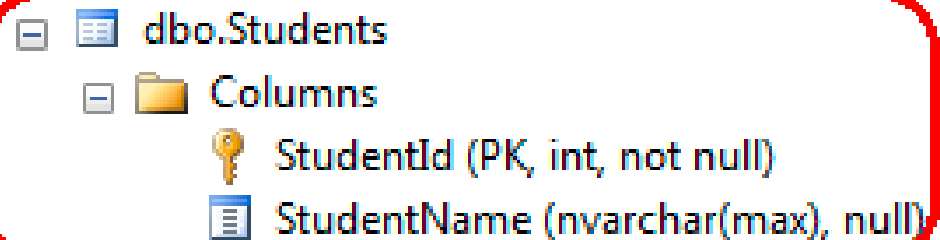
Klucze proste tworzone z pola typu Integer tworzone są jako pola **Identity** klucze złożone **nie mają** ustawionej tej właściwości

Domyślnie EF mapuje do bazy danych wszystkie property klasy, dla których zdefiniowane są gettery i settery.

Jeżeli chcemy **pomiąć** mapowanie jakiejś właściwości możemy albo adnotować ją jako **NotMapped** albo pomiąć przy jej definicji **getter** bądź **setter**

```
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }

    [NotMapped]
    public int Age { get; set; }
}
```

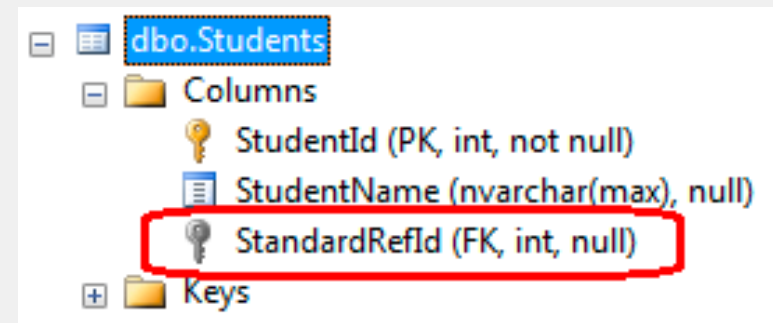


Domyślnie zachowanie możemy zmienić adnotacją **ForeignKey**

Adnotacji ForeignKey możemy użyć dla właściwości **przewidzianej jako klucz obcy w encji zależnej** (ma sens jeżeli nazwa właściwości nie pozwoli „zadziałać” konwencji)

```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }

    [ForeignKey("Standard")]
    public int StandardRefId { get; set; }
    public Standard Standard { get; set; }
}
```





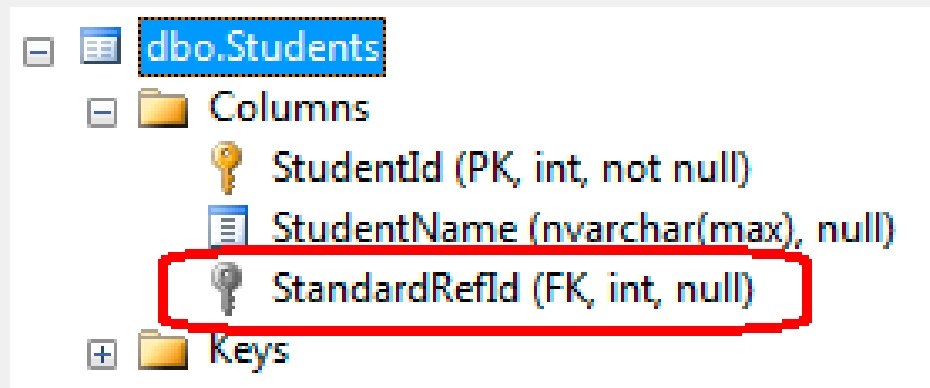
Domyślnie zachowanie możemy zmienić adnotacją **ForeignKey**

Adnotacji **ForeignKey** możemy użyć dla **navigation property w encji zależnej** (jeśli chcemy aby klucz obcy miał nazwę inna niż zgodna z konwencją)

```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }

    public int StandardRefId { get; set; }

    [ForeignKey("StandardRefId")]
    public Standard Standard { get; set; }
}
```



Domyślnie zachowanie możemy zmienić adnotacją **ForeignKey**

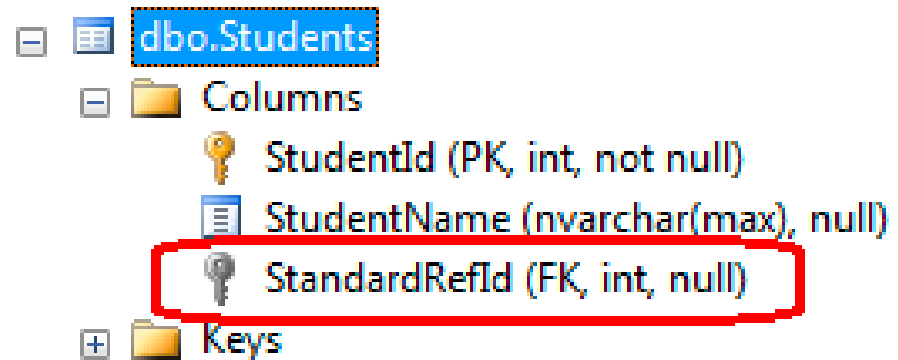
Adnotacji **ForeignKey** możemy użyć dla **navigation property** w encji **podstawowej**

```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }

    public int StandardRefId { get; set; }
    public Standard Standard { get; set; }
}

public class Standard
{
    public int StandardId { get; set; }
    public string StandardName { get; set; }

    [ForeignKey("StandardRefId")]
    public ICollection<Student> Students { get; set; }
}
```



Adnotacją **Index** możemy założyć index na konkretnym atrybucie encji

```
class Student
{
    public int Student_ID { get; set; }
    public string StudentName { get; set; }

    [Index]
    public int RegistrationNumber { get; set; }
}
```

Domyślnie Indeks otrzymuje nazwę **IX\_{property name}**. Oczywiście, może to zostać zmienione:

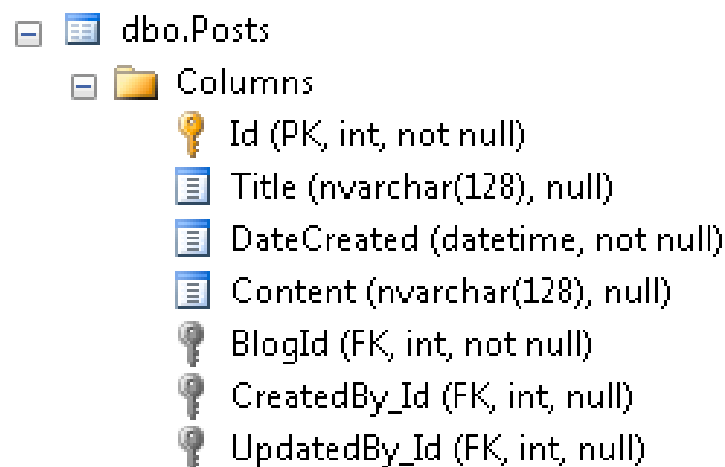
```
[Index( "INDEX_REGNUM", IsClustered=true, IsUnique=true )]
public int RegistrationNumber { get; set; }
```

Konwencję przyjętą w EF **nie radzą** sobie w sytuacji jeśli pomiędzy encjami występuje **wielokrotna** relacja. Aby to zamodelować poprawnie używamy adnotacji **InverseProperty**

```
public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public DateTime DateCreated { get; set; }
    public string Content { get; set; }
    public int BlogId { get; set; }
    [ForeignKey("BlogId")]
    public Blog Blog { get; set; }
    public ICollection<Comment> Comments { get; set; }
}
```

```
[InverseProperty("CreatedBy")]
public List<Post> PostsWritten { get; set; }

[InverseProperty("UpdatedBy")]
public List<Post> PostsUpdated { get; set; }
```

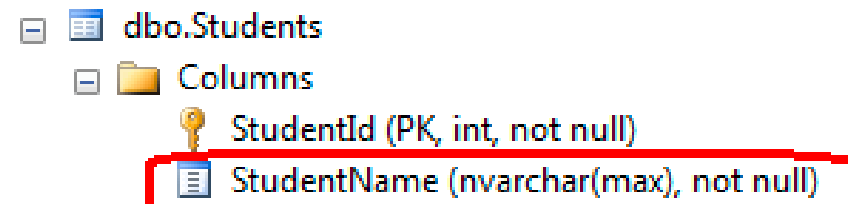


dbo.Posts

- Columns
  - Id (PK, int, not null)
  - Title (nvarchar(128), null)
  - DateCreated (datetime, not null)
  - Content (nvarchar(128), null)
  - BlogId (FK, int, not null)
  - CreatedBy\_Id (FK, int, null)
  - UpdatedBy\_Id (FK, int, null)

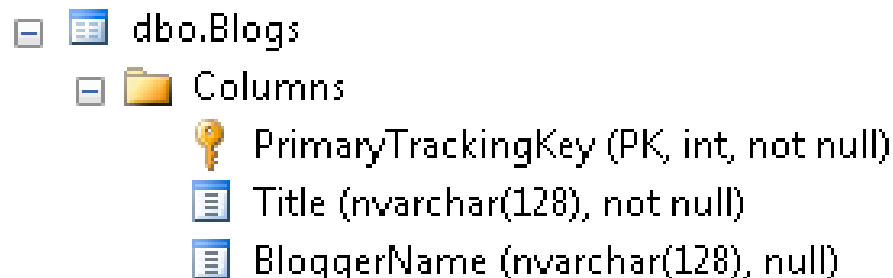
Adnotacją **Required** możemy wymusić „obowiązkowość” wartości w danym polu (czyli ustawienie flagi **not null** na kolumnie)

```
public class Student
{
    public int StudentID { get; set; }
    [Required]
    public string StudentName { get; set; }
}
```



Adnotacjami **MaxLength** oraz **MinLength** możemy wymusić minimalną i maksymalną długość wartości w odpowiadającej kolumnie typu string (varchar) oraz byte[]

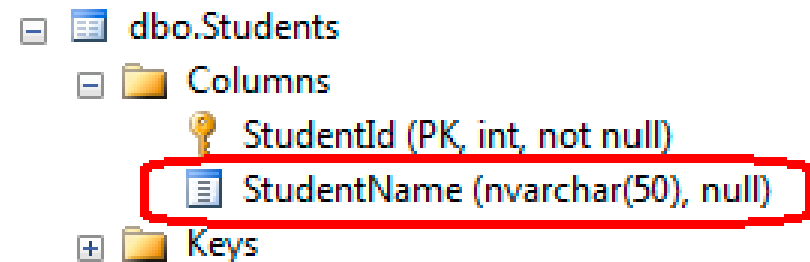
```
[MaxLength(10, ErrorMessage="BloggerName must be 10 characters or less"),MinLength(5)]  
public string BloggerName { get; set; }
```



```
dbo.Blogs  
└─ Columns  
    └─ PrimaryTrackingKey (PK, int, not null)  
    └─ Title (nvarchar(128), not null)  
    └─ BloggerName (nvarchar(128), null)
```

Adnotacją **StringLength** możemy nałożyć ograniczenie na długość pola **string (varchar)**

```
public class Student
{
    public int StudentID { get; set; }
    [StringLength(50)]
    public string StudentName { get; set; }
}
```

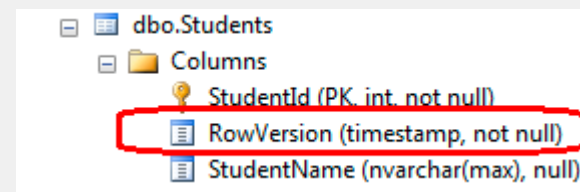


Adnotacją **TimeStamp** możemy „założyć” kolumnę typu TimeStamp. W każdej encji może być **tylko jeden** atrybut adnotowany w ten sposób.

EF automatycznie wykorzystuje to pole przy operacji **update** sprawdzając czy nie próbujemy updatować „nie swojej” wartości

```
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }

    [Timestamp]
    public byte[] RowVersion { get; set; }
}
```





Adnotacją **ConcurrencyCheck** możemy adnotować **dowolną** liczbę atrybutów **dowolnego typu** jeśli chcemy aby ich wartość była sprawdzana przed wykonaniem operacji **update**

```
public class Student
{
    public int StudentId { get; set; }

    [ConcurrencyCheck]
    public string StudentName { get; set; }
}
```

```
using(var context = new SchoolContext())
{
    var std = new Student()
    {
        StudentName = "Bill"
    };

    context.Students.Add(std);
    context.SaveChanges();

    std.StudentName = "Steve";
    context.SaveChanges();
}
```

```
exec sp_executesql N'UPDATE [dbo].[Students]
SET [StudentName] = @0
WHERE ((([StudentId] = @1) AND ([StudentName] = @2))
',N'@0 nvarchar(max) ,@1 int,@2 nvarchar(max) ',@0=N'Steve',@1=1,@2=N'Bill'
go
```

# Relacja One-2-Zero-or-One

Relację **One-2-One** (a właściwie One-2-Zero-or-One) uzyskamy jeśli klucz główny jednej tabeli staje się jednocześnie kluczem obcym do innej tabeli. W EF Uzyskamy to odpowiednimi adnotacjami

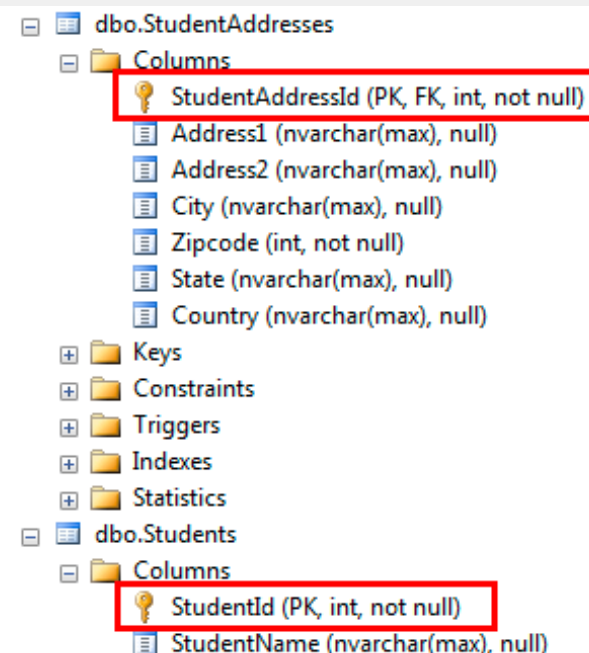
```
public class StudentAddress
{
    [ForeignKey("Student")]
    public int StudentAddressId { get; set; }
```

```
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }

    public virtual StudentAddress Address { get; set; }
}

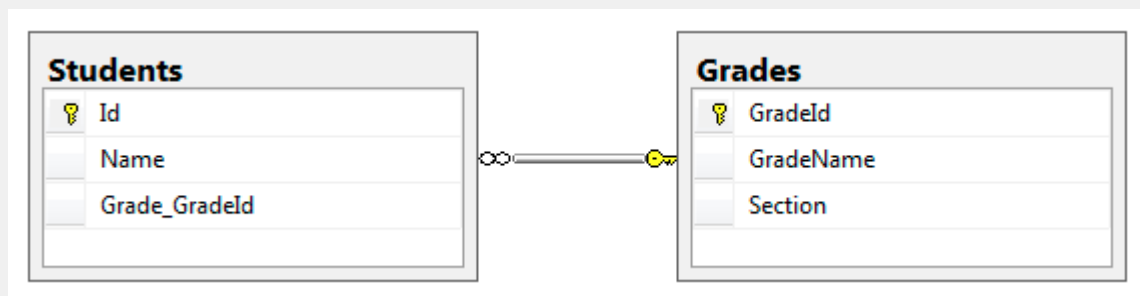
public class StudentAddress
{
    public int StudentAddressId { get; set; }
    public string Address1 { get; set; }
    public string Address2 { get; set; }
    public string City { get; set; }
    public int Zipcode { get; set; }
    public string State { get; set; }
    public string Country { get; set; }

    public virtual Student Student { get; set; }
}
```



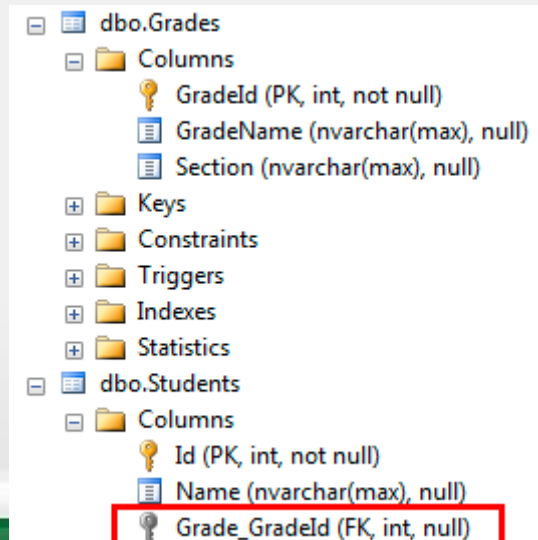
# Relacja Many-2-One

Jednokierunkową relację **Many-2-One** zamodelujemy w EF wstawiając do encji po stronie „many” property „wskazującą” na inną encję strony „one”



```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Grade Grade { get; set; }
}

public class Grade
{
    public int GradeId { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }
}
```



Jednokierunkową relację **One-2-Many** zamodelujemy w EF wstawiając do encji strony „one” kolekcję obiektów encji strony „many”

```
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }
}

public class Grade
{
    public int GradeId { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }

    public ICollection<Student> Students { get; set; }
}
```

Dwukierunkową relację **1:n-n:1** zamodelujemy w EF łącząc oba poprzednie rozwiązania

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Grade Grade { get; set; }
}

public class Grade
{
    public int GradeID { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }

    public ICollection<Student> Student { get; set; }
}
```

# Relacja Many-2-Many

Dwukierunkową relację **Many-2-Many** zamodelujemy w EF wstawiając w obu encjach kolekcje obiektów encji powiązanej

```
public class Student
{
    public Student()
    {
        this.Courses = new HashSet<Course>();
    }

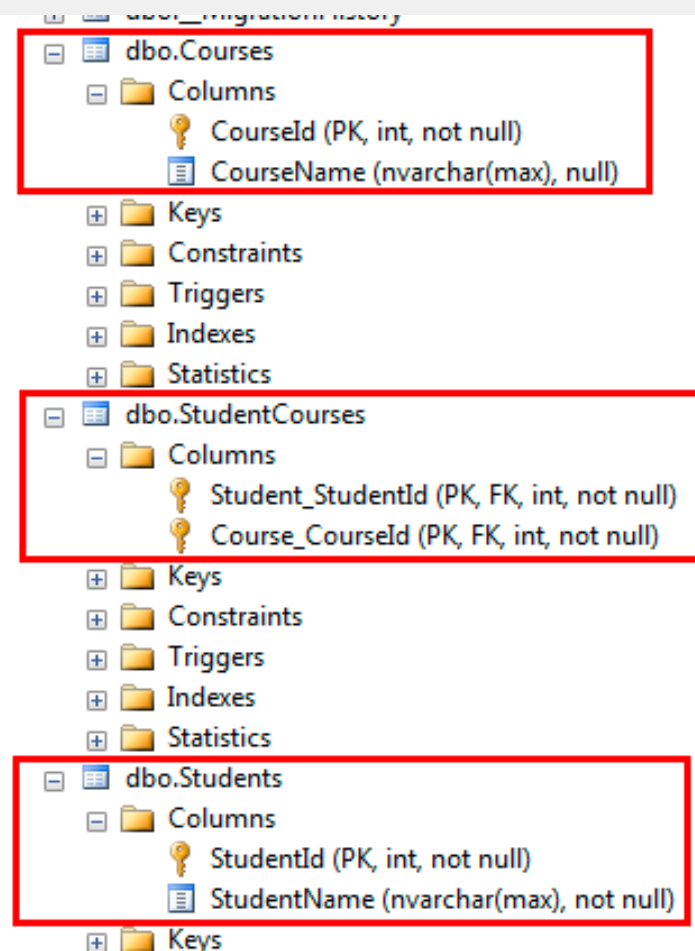
    public int StudentId { get; set; }
    [Required]
    public string StudentName { get; set; }

    public virtual ICollection<Course> Courses { get; set; }
}

public class Course
{
    public Course()
    {
        this.Students = new HashSet<Student>();
    }

    public int CourseId { get; set; }
    public string CourseName { get; set; }

    public virtual ICollection<Student> Students { get; set; }
}
```



## LINQ - *.NET Language-Integrated Query*

- Zestaw deklaratywnych zapytań ogólnego przeznaczenia, służących do dostępu do różnych źródeł danych z aplikacji obiektowych.

- Technologia LINQ jest dostępna na platformie Microsoft .Net Framework od wersji 3.5
- Dostępna dla języków C# i Visual Basic.Net



Język: C#

Język: VB

Inne języki...

.Net Language Integrated Query (LINQ)

Źródła danych LINQ

LINQ To  
Objects

LINQ To  
DataSets

LINQ To  
SQL

LINQ To  
Entities

LINQ To  
XML

Inne...



Objects



Relational

```
<book>  
<title/>  
<author/>  
<price/>  
</book>
```

XML

- LINQ2Entities – framework umożliwiający zadawanie zapytań Language-Integrated Query (LINQ) do źródeł/modeli danych opartych o Entity Framework
- LINQ2Entities konwertuje zapytania Language-Integrated Queries (LINQ) do zrozumiałej przez Entity Framework postaci drzewa komend, umożliwia wykonanie takiego zapytania na modelu EntityFramework'owym i odpowiada za zwrócenie obiektów rezultatu które mogą być wykorzystane zarówno w ramach samego EF jak i LINQ

- W L2E zapytania mogą być definiowane w oparciu o:
  - query expression syntax
  - method-based syntax
  - Query expression syntax to novum wprowadzone w C# 3.0 i VB 9.0 i składa się ze zbioru „klauszul” zapisywanych w sposób deklaratywny przypominający Transact-SQL czy XQuery.
  - Niestety CLR „nie rozumie” bezpośrednio zapytań opartych o query expression syntax.
  - Dlatego w czasie kompilacji zapytania takie są tłumaczone do czegoś zrozumiałego dla CLR – czyli do wywołań metod.
  - Metody te nazywane są standardowymi operatorami zapytań
  - Deweloper ma wybór pomiędzy konstrukcją zapytań w oparciu o query syntax, a bezpośrednim wywoływaniem rozumianych przez CLR metod

# Query synthax vs method syntax



- Query syntax i method syntax są semantycznie tożsame aczkolwiek wiele osób uważa składnie query syntax jako łatwiejszą i bardziej zrozumiałą (w konstrukcji i odczycie)
- Niestety niektóre zapytania **muszą** być wyrażone w postaci wywołań metod (method syntax). W szczególności są to wywołania funkcji agregujących typu:
  - Podaj liczbę elementów spełniających określony warunek
  - Podaj maksymalną/minimalną wartość w kolekcji źródłowej

# Query syntax vs method syntax

```
class QueryVMethodSyntax
{
    static void Main()
    {
        int[] numbers = { 5, 10, 8, 3, 6, 12};

        //Query syntax:
        IEnumerable<int> numQuery1 =
            from num in numbers
            where num % 2 == 0
            orderby num
            select num;

        //Method syntax:
        IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n);

        foreach (int i in numQuery1)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine(System.Environment.NewLine);
        foreach (int i in numQuery2)
        {
            Console.Write(i + " ");
        }

        // Keep the console open in debug mode.
        Console.WriteLine(System.Environment.NewLine);
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

# Query expression syntax

# Data Source (From)

- W zapytaniu LINQ (query syntax) pierwszy krok to specyfikacja źródła danych. Dlatego w tych zapytaniach klauzula FROM następuje zawsze jako pierwsza (to determinuje np. typy, operatory etc).
- ```
Dim query = From cust In customers
            .
```
- W powyższym przykładzie klauzula from specyfikuje „customers” jako źródło danych oraz tzw. zmienną zakresu cust
  - Zmienna zakresu (range variable) przypomina zmienną kontrolną pętli
  - Kiedy zapytanie jest wykonywane (zwyczajowo z wykorzystaniem pętli **For Each** zmienna zakresu jest wykorzystywana jako referencja do kolejnych elementów kolekcji customers
  - Ponieważ kompilator jest w stanie wywnioskować typ zmiennej cust nie musi być on explicite deklarowany

- Klauzula from może zostać użyta w zapytaniu więcej niż raz jeśli chcemy wyspecyfikować różne kolekcje które mają być połączone
- W takim przypadku podczas egezkucji zapytania każda kolekcja
  - Może być iterowana niezależnie
  - Mogą być połączone jeśli istnieje pomiędzy nimi relacja
- Kolekcje mogą zostać połączone
  - implicite (klauzula select)
  - Explicite (klauzule Join i GroupJoin)
- Alternatywnie można zdefiniować wiele zmiennych zakresu i kolekcji w pojedynczej klauzuli FROM odseparowanych przecinkami

```
' Multiple From clauses in a query.  
Dim result = From var1 In collection1, var2 In collection2  
  
' Equivalent syntax with a single From clause.  
Dim result2 = From var1 In collection1  
               From var2 In collection2
```



- Klauzula from definiuje coś w rodzaju „zasięgu” zapytania (query scope) (przypominający zasięg pętli)
- Dlatego każda zmienna zakresu musi mieć w zapytaniu unikalną nazwę
- Następujące po sobie klauzule FROM mogą odnosić się albo do zmiennej zakresu danej klauzuli FROM albo do zmiennej (zmiennych) zakresu poprzedniej klauzuli from

```
Dim allOrders = From cust In GetCustomerList()  
                From ord In cust.Orders  
                Select ord
```

- Kolekcja produktów....

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Product> productsQuery = from product in context.Products
   select product;

    Console.WriteLine("Product Names:");
    foreach (var prod in productsQuery)
    {
        Console.WriteLine(prod.Name);
    }
}
```

- Kolekcja nazw produktów

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<string> productNames =
        from p in context.Products
        select p.Name;

    Console.WriteLine("Product Names:");
    foreach (String productName in productNames)
    {
        Console.WriteLine(productName);
    }
}
```

- Projekcja do kolekcji typów anonimowych...

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query =
        from product in context.Products
        select new
        {
            ProductId = product.ProductID,
            ProductName = product.Name
        };

    Console.WriteLine("Product Info:");
    foreach (var productInfo in query)
    {
        Console.WriteLine("Product Id: {0} Product name: {1} ",
            productInfo.ProductId, productInfo.ProductName);
    }
}
```

- Wszystkie zamówienia „online”...

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var onlineOrders =
        from order in context.SalesOrderHeaders
        where order.OnlineOrderFlag == true
        select new
        {
            SalesOrderID = order.SalesOrderID,
            OrderDate = order.OrderDate,
            SalesOrderNumber = order.SalesOrderNumber
        };

    foreach (var onlineOrder in onlineOrders)
    {
        Console.WriteLine("Order ID: {0} Order date: {1:d} Order number: {2}",
            onlineOrder.SalesOrderID,
            onlineOrder.OrderDate,
            onlineOrder.SalesOrderNumber);
    }
}
```

- Zamówienia o ilości z przedziału od 2 do 6

```
int orderQtyMin = 2;
int orderQtyMax = 6;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query =
        from order in context.SalesOrderDetails
        where order.OrderQty > orderQtyMin && order.OrderQty < orderQtyMax
        select new
        {
            SalesOrderID = order.SalesOrderID,
            OrderQty = order.OrderQty
        };

    foreach (var order in query)
    {
        Console.WriteLine("Order ID: {0} Order quantity: {1}",
            order.SalesOrderID, order.OrderQty);
    }
}
```

- **Where...Contains** do pobrania produktów o wartościach *ProductModelID* zgodnych z podanymi w tablicy

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    int?[] productModelIds = {19, 26, 118};
    var products = from p in AWEntities.Products
                   where productModelIds.Contains(p.ProductModelID)
                   select p;
    foreach (var product in products)
    {
        Console.WriteLine("{0}: {1}", product.ProductModelID, product.ProductID);
    }
}
```

- Uwaga: Jako część klauzuli **Where...Contains** można użyć typów Array, List lub dowolnej kolekcji implementującej interface IEnumerable

- Where...Contains z kolekcjami inline'owymi

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    var products = from p in AWEntities.Products
                    where (new int?[] { 19, 26, 18 }).Contains(p.ProductModelID) ||
                           (new string[] { "L", "XL" }).Contains(p.Size)
                    select p;
    foreach (var product in products)
    {
        Console.WriteLine("{0}: {1}, {2}", product.ProductID,
                               product.ProductModelID,
                               product.Size);
    }
}
```



- Kontakty posortowane po nazwisku...

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Contact> sortedNames =
        from n in context.Contacts
        orderby n.LastName
        select n;

    Console.WriteLine("The sorted list of last names:");
    foreach (Contact n in sortedNames)
    {
        Console.WriteLine(n.LastName);
    }
}
```

- Kontakty posortowane po długości nazwiska...

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Contact> sortedNames =
        from n in context.Contacts
        orderby n.LastName.Length
        select n;

    Console.WriteLine("The sorted list of last names (by length):");
    foreach (Contact n in sortedNames)
    {
        Console.WriteLine(n.LastName);
    }
}
```

- Produkty posortowane malejąco po cenie

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Decimal> sortedPrices =
        from p in context.Products
        orderby p.ListPrice descending
        select p.ListPrice;

    Console.WriteLine("The list price from highest to lowest:");
    foreach (Decimal price in sortedPrices)
    {
        Console.WriteLine(price);
    }
}
```

- Kontakty posortowane po nazwisku, następnie po imieniu....

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Contact> sortedContacts =
        from contact in context.Contacts
        orderby contact.LastName, contact.FirstName
        select contact;

    Console.WriteLine("The list of contacts sorted by last name then by first name:");
    foreach (Contact sortedContact in sortedContacts)
    {
        Console.WriteLine(sortedContact.LastName + ", " + sortedContact.FirstName);
    }
}
```

- Produkty posortowane rosnąco po nazwie a malejąco po cenie

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Product> query =
        from product in context.Products
        orderby product.Name, product.ListPrice descending
        select product;

    foreach (Product product in query)
    {
        Console.WriteLine("Product ID: {0} Product Name: {1} List Price {2}",
            product.ProductID,
            product.Name,
            product.ListPrice);
    }
}
```

- Pobranie ContactID i ilości jego zamówień...

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;

    //Can't find field SalesOrderContact
    var query =
        from contact in contacts
        select new
        {
            CustomerID = contact.ContactID,
            OrderCount = contact.SalesOrderHeaders.Count()
        };

    foreach (var contact in query)
    {
        Console.WriteLine("CustomerID = {0} \t OrderCount = {1}",
            contact.CustomerID,
            contact.OrderCount);
    }
}
```

- Suma kwoty do zapłaty dla każdego ContactID

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        select new
        {
            Category = g.Key,
            TotalDue = g.Sum(order => order.TotalDue)
        };

    foreach (var order in query)
    {
        Console.WriteLine("ContactID = {0} \t TotalDue sum = {1}",
            order.Category, order.TotalDue);
    }
}
```

- Pobranie wszystkich adresów z Seattle za wyjątkiem dwóch pierwszych

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Address> addresses = context.Addresses;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    //LINQ to Entities only supports Skip on ordered collections.
    var query = (
        from address in addresses
        from order in orders
        where address.AddressID == order.Address.AddressID
            && address.City == "Seattle"
        orderby order.SalesOrderID
        select new
        {
            City = address.City,
            OrderID = order.SalesOrderID,
            OrderDate = order.OrderDate
        }).Skip(2);

    Console.WriteLine("All but first 2 orders in Seattle:");
    foreach (var order in query)
    {
        Console.WriteLine("City: {0} Order ID: {1} Total Due: {2:d}",
            order.City, order.OrderID, order.OrderDate);
    }
}
```



## ■ Pobranie pierwszych trzech adresów z Seattle

```
String city = "Seattle";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Address> addresses = context.Addresses;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query = (
        from address in addresses
        from order in orders
        where address.AddressID == order.Address.AddressID
            && address.City == city
        select new
        {
            City = address.City,
            OrderID = order.SalesOrderID,
            OrderDate = order.OrderDate
        }).Take(3);
    Console.WriteLine("First 3 orders in Seattle:");
    foreach (var order in query)
    {
        Console.WriteLine("City: {0} Order ID: {1} Total Due: {2:d}",
            order.City, order.OrderID, order.OrderDate);
    }
}
```

- Operacja join pomiędzy tabelami (kolekcjami) SalesOrderHeader oraz SalesOrderDetail i pobranie wszystkich onlinowych zamówień z sierpnia

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;
    ObjectSet<SalesOrderDetail> details = context.SalesOrderDetails;

    var query =
        from order in orders
        join detail in details
        on order.SalesOrderID equals detail.SalesOrderID
        where order.OnlineOrderFlag == true
        && order.OrderDate.Month == 8
        select new
        {
            SalesOrderID = order.SalesOrderID,
            SalesOrderDetailID = detail.SalesOrderDetailID,
            OrderDate = order.OrderDate,
            ProductID = detail.ProductID
        };

    foreach (var order in query)
    {
        Console.WriteLine("{0}\t{1}\t{2:d}\t{3}",
            order.SalesOrderID,
            order.SalesOrderDetailID,
            order.OrderDate,
            order.ProductID);
    }
}
```

# GroupJoin



- Pobranie
- (ilości) zamówień dla każdego kontaktu

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from contact in contacts
        join order in orders
        on contact.ContactID
        equals order.Contact.ContactID into contactGroup
        select new
        {
            ContactID = contact.ContactID,
            OrderCount = contactGroup.Count(),
            Orders = contactGroup
        };

    foreach (var group in query)
    {
        Console.WriteLine("ContactID: {0}", group.ContactID);
        Console.WriteLine("Order count: {0}", group.OrderCount);
        foreach (var orderInfo in group.Orders)
        {
            Console.WriteLine("    Sale ID: {0}", orderInfo.SalesOrderID);
        }
        Console.WriteLine("");
    }
}
```

- Zwrócenie pierwszego kontaktu dla którego imie to "Brooke"

```
string firstName = "Brooke";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;

    Contact query = (
        from contact in contacts
        where contact.FirstName == firstName
        select contact)
        .First();

    Console.WriteLine("ContactID: " + query.ContactID);
    Console.WriteLine("FirstName: " + query.FirstName);
    Console.WriteLine("LastName: " + query.LastName);
}
```

- Adresy pogrupowane po kodzie pocztowym, zwrócone jako typ anonimowy

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query =
        from address in context.Addresses
        group address by address.PostalCode into addressGroup
        select new { PostalCode = addressGroup.Key,
                    AddressLine = addressGroup };

    foreach (var addressGroup in query)
    {
        Console.WriteLine("Postal Code: {0}", addressGroup.PostalCode);
        foreach (var address in addressGroup.AddressLine)
        {
            Console.WriteLine("\t" + address.AddressLine1 +
                              address.AddressLine2);
        }
    }
}
```

- Kontakty pogrupowane po pierwszej literze nazwiska, dodatkowo posortowane i zwrócone jako typ anonimowy

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query = (
        from contact in context.Contacts
        group contact by contact.LastName.Substring(0, 1) into contactGroup
        select new { FirstLetter = contactGroup.Key, Names = contactGroup }).
        OrderBy(letter => letter.FirstLetter);

    foreach (var contact in query)
    {
        Console.WriteLine("Last names that start with the letter '{0}':",
            contact.FirstLetter);
        foreach (var name in contact.Names)
        {
            Console.WriteLine(name.LastName);
        }
    }
}
```

- Zamówienia pogrupowane po kliencie wraz z ilością zamówień w grupie

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query = from order in context.SalesOrderHeaders
                group order by order.CustomerID into idGroup
                select new {CustomerID = idGroup.Key,
                           OrderCount = idGroup.Count(),
                           Sales = idGroup};

    foreach (var orderGroup in query)
    {
        Console.WriteLine("Customer ID: {0}", orderGroup.CustomerID);
        Console.WriteLine("Order Count: {0}", orderGroup.OrderCount);

        foreach (SalesOrderHeader sale in orderGroup.Sales)
        {
            Console.WriteLine("    Sale ID: {0}", sale.SalesOrderID);
        }

        Console.WriteLine("");
    }
}
```

- Navigation properties w EF pozwalają zlokalizować encje „na końcu asocjacji”
- Pozwalają także nawigować po encjach zgodnie z łańcuchem asocjacji



- Wyciągamy zamówienia i następnie korzystamy z „Navigation property” **order.SalesOrderDetail** żeby pokazać szczegóły takich zamówień

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<SalesOrderHeader> query =
        from order in context.SalesOrderHeaders
        where order.OrderDate >= new DateTime(2003, 12, 1)
        select order;

    Console.WriteLine("Orders that were made after December 1, 2003:");
    foreach (SalesOrderHeader order in query)
    {
        Console.WriteLine("OrderID {0} Order date: {1:d} ",
            order.SalesOrderID, order.OrderDate);
        foreach (SalesOrderDetail orderDetail in order.SalesOrderDetails)
        {
            Console.WriteLine("  Product ID: {0} Unit Price {1}",
                orderDetail.ProductID, orderDetail.UnitPrice);
        }
    }
}
```

- ID Kontaktu i suma płatności dla kontaktu „Zhou”. Uwaga na **Contact.SalesOrderHeader**

```
string lastName = "Zhou";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;

    var ordersQuery = from contact in contacts
                      where contact.LastName == lastName
                      select new
                      {
                          ContactID = contact.ContactID,
                          Total = contact.SalesOrderHeaders.Sum(o => o.TotalDue)
                      };

    foreach (var contact in ordersQuery)
    {
        Console.WriteLine("Contact ID: {0} Orders total: {1}", contact.ContactID, contact.Total);
    }
}
```

- Wszystkie zamówienia dla klienta "Zhou". Uwaga na **Contact.SalesOrderHeader** navigation property

```
string lastName = "Zhou";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;

    var ordersQuery = from contact in contacts
                      where contact.LastName == lastName
                      select new { LastName = contact.LastName, Orders = contact.SalesOrderHeaders };

    foreach (var order in ordersQuery)
    {
        Console.WriteLine("Name: {0}", order.LastName);
        foreach (SalesOrderHeader orderInfo in order.Orders)
        {
            Console.WriteLine("Order ID: {0}, Order date: {1}, Total Due: {2}",
                              orderInfo.SalesOrderID, orderInfo.OrderDate, orderInfo.TotalDue);
        }
        Console.WriteLine("");
    }
}
```

- Wszystkie zamówienia złożone po December 1, 2003 Uwaga na **order.SalesOrderDetail** navigation property

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<SalesOrderHeader> query =
        from order in context.SalesOrderHeaders
        where order.OrderDate >= new DateTime(2003, 12, 1)
        select order;

    Console.WriteLine("Orders that were made after December 1, 2003:");
    foreach (SalesOrderHeader order in query)
    {
        Console.WriteLine("OrderID {0} Order date: {1:d} ",
            order.SalesOrderID, order.OrderDate);
        foreach (SalesOrderDetail orderDetail in order.SalesOrderDetails)
        {
            Console.WriteLine(" Product ID: {0} Unit Price {1}",
                orderDetail.ProductID, orderDetail.UnitPrice);
        }
    }
}
```

# Wyrażenia lambda

# Wykorzystanie wyrażeń lambda



- Druga z możliwości komponowania zapytań w technologii L2E to tzw. method-based queries.
- Polega na bezpośrednim wywoływaniu metod „operatorowych” LINQ przekazując **wyrażenia lambda** jako parametry

# Wyrażenia lambda



- Wyrażenia postaci  $(\text{num} \Rightarrow \text{num} \% 2 == 0)$  noszą nazwę wyrażen lambda
- W C# operator  $\Rightarrow$  to tzw operator lambda („przechodzi w”)
- Lewa strona to zmienna wejściowa
- Ponieważ (w tym przypadku) kompilator jest w stanie „wywnioskować” typ zmiennej num nie trzeba go deklarować explicite
- „Ciało” wyrażenia lambda to typowe wyrażenia mogące zawierać stałe, zmienne, wywołania metod i wyrażenia logiczne
- Wartością zwracaną jest wartość wyrażenia

- Wyrażenia lambda mogą zostać przypisane do typu „delegatowego”

```
delegate int del(int i);  
static void Main(string[] args)  
{  
    del myDelegate = x => x * x;  
    int j = myDelegate(5); //j = 25  
}
```

- Lub do typu Expression:

```
using System.Linq.Expressions;  
  
namespace ConsoleApplication1  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Expression<del> myET = x => x * x;  
        }  
    }  
}
```

- Operator  $\Rightarrow$  jest operatorem prawostronnym z takimi samymi zasadami priorytetowania i precedencji jak operator przypisania ( $=$ )



# Wyrażenia lambda a LINQ



- Podczas tworzenia zapytań LINQ na kolekcjach zgodnych z [Enumerable](#) przekazywany paramet to delegat postaci [System.Func<T, TResult>](#)
- Wyrażenia lambda są najwygodniejszym sposobem tworzenia delegatu
- Kiedy tworzone są zapytania LINQ na kolekcjach zgodnych z [System.Linq.Queryable](#) wtedy parametry są typu [System.Linq.Expressions.Expression<Func>](#)
- Gdzie Func to dowolny delegat przyjmujący do 16 parametrów wejściowych
- Ponownie – wyrażenie lambda jest wygodnym sposobem

na stworzenie tego typu drzewa wyrażen

- Postać ogólna

```
(input parameters) => expression
```

- Nawiasy opcjonalne przy pojedynczym parametrze wejściowym – w innym przypadku obowiązkowe. Parametry wejściowe oddzielamy przecinkami

```
(x, y) => x == y
```

- Czasami kompilator nie może wywnioskować typu parametru (bądź może to być niejednoznaczne). W takich przypadkach możemy zadeklarować taki typ explicite

```
(int x, string s) => s.Length > x
```

- Wyrażenie lambda może zawierać wywołania metod

```
() => SomeMethod()
```

- Wyrażenia lambda mogą składać się z wielu „operacji” (choć zwykle nie więcej niż dwie - trzy).
- Umieszczamy je w klamrach i oddzielamy średnikami

```
delegate void TestDelegate(string s);  
...  
TestDelegate myDel = n => { string s = n + " " + "World"; Console.WriteLine(s); };  
myDel("Hello");
```

- Wyrażenia lambda mogą być przetwarzane asynchronicznie (słowa kluczowe **async** i **await**)

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            // ExampleMethodAsync returns a Task.
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event handler.\r\n";
        };
    }

    async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

# Method based syntax

- Pobranie sekwencji składającej się z nazw produktów

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<string> productNames = context.Products
        .Select(p => p.Name);

    Console.WriteLine("Product Names:");
    foreach (String productName in productNames)
    {
        Console.WriteLine(productName);
    }
}
```

- Pobranie nazwy i ID produktu w postaci typu anonimowego

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query = context.Products
        .Select(product => new
        {
            ProductId = product.ProductID,
            ProductName = product.Name
        });

    Console.WriteLine("Product Info:");
    foreach (var productInfo in query)
    {
        Console.WriteLine("Product Id: {0} Product name: {1} ",
            productInfo.ProductId, productInfo.ProductName);
    }
}
```



- Wywołanie metody **SelectMany** do pobrania wszystkich zamówień o wartości mniejszej od 500.00.

```
decimal totalDue = 500.00M;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
    contacts.SelectMany(
        contact => orders.Where(order =>
            (contact.ContactID == order.Contact.ContactID)
            && order.TotalDue < totalDue)
            .Select(order => new
            {
                ContactID = contact.ContactID,
                LastName = contact.LastName,
                FirstName = contact.FirstName,
                OrderID = order.SalesOrderID,
                Total = order.TotalDue
            }
        ));

    foreach (var smallOrder in query)
    {
        Console.WriteLine("Contact ID: {0} Name: {1}, {2} Order ID: {3} Total Due: ${4} ",
            smallOrder.ContactID, smallOrder.LastName, smallOrder.FirstName,
            smallOrder.OrderID, smallOrder.Total);
    }
}
```

- Wywołanie **SelectMany** do pobrania zamówień późniejszych niż October 1, 2002

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        contacts.SelectMany(
            contact => orders.Where(order =>
                (contact.ContactID == order.Contact.ContactID)
                && order.OrderDate >= new DateTime(2002, 10, 1))
                .Select(order => new
                {
                    ContactID = contact.ContactID,
                    LastName = contact.LastName,
                    FirstName = contact.FirstName,
                    OrderID = order.SalesOrderID,
                    OrderDate = order.OrderDate
                }
            ));

    foreach (var order in query)
    {
        Console.WriteLine("Contact ID: {0} Name: {1}, {2} Order ID: {3} Order date: {4:d} ",
            order.ContactID, order.LastName, order.FirstName,
            order.OrderID, order.OrderDate);
    }
}
```

- Wszystkie „onlinowe” zamówienia....

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var onlineOrders = context.SalesOrderHeaders
        .Where(order => order.OnlineOrderFlag == true)
        .Select(s => new { s.SalesOrderID, s.OrderDate, s.SalesOrderNumber });

    foreach (var onlineOrder in onlineOrders)
    {
        Console.WriteLine("Order ID: {0} Order date: {1:d} Order number: {2}",
            onlineOrder.SalesOrderID,
            onlineOrder.OrderDate,
            onlineOrder.SalesOrderNumber);
    }
}
```

- Pobranie zamówień o ilości spomiędzy 2 a 6

```
int orderQtyMin = 2;
int orderQtyMax = 6;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query = context.SalesOrderDetails
        .Where(order => order.OrderQty > orderQtyMin && order.OrderQty < orderQtyMax)
        .Select(s => new { s.SalesOrderID, s.OrderQty });

    foreach (var order in query)
    {
        Console.WriteLine("Order ID: {0} Order quantity: {1}",
            order.SalesOrderID, order.OrderQty);
    }
}
```

- Wszystkie czerwone kolory....

```
String color = "Red";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query = context.Products
        .Where(product => product.Color == color)
        .Select(p => new { p.Name, p.ProductNumber, p.ListPrice });

    foreach (var product in query)
    {
        Console.WriteLine("Name: {0}", product.Name);
        Console.WriteLine("Product number: {0}", product.ProductNumber);
        Console.WriteLine("List price: ${0}", product.ListPrice);
        Console.WriteLine("");
    }
}
```

- Wszystkie produkty o *ProductModelID* pasującym do wartości z tablicy

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    int?[] productModelIds = { 19, 26, 118 };
    var products = AWEntities.Products.
        Where(p => productModelIds.Contains(p.ProductModelID));

    foreach (var product in products)
    {
        Console.WriteLine("{0}: {1}", product.ProductModelID, product.ProductID);
    }
}
```

- WhereContains z kolekcjami inlinowymi

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    var products = AWEntities.Products.
        Where(p => (new int?[] { 19, 26, 18 }).Contains(p.ProductModelID) ||
            (new string[] { "L", "XL" }).Contains(p.Size));

    foreach (var product in products)
    {
        Console.WriteLine("{0}: {1}, {2}", product.ProductID,
            product.ProductModelID,
            product.Size);
    }
}
```



- Kontakty posortowane po nazwisku a następnie po imieniu

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Contact> sortedContacts = context.Contacts
        .OrderBy(c => c.LastName)
        .ThenBy(c => c.FirstName);

    Console.WriteLine("The list of contacts sorted by last name then by first name:");
    foreach (Contact sortedContact in sortedContacts)
    {
        Console.WriteLine(sortedContact.LastName + ", " + sortedContact.FirstName);
    }
}
```



# OrderBy...ThenByDescending

- Produkty posortowane po cenie, a w ramach ceny malejąco po nazwie

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IOrderedQueryable<Product> query = context.Products
        .OrderBy(product => product.ListPrice)
        .ThenByDescending(product => product.Name);

    foreach (Product product in query)
    {
        Console.WriteLine("Product ID: {0} Product Name: {1} List Price {2}",
            product.ProductID,
            product.Name,
            product.ListPrice);
    }
}
```

- Średnia cena produktów....

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Product> products = context.Products;

    Decimal averageListPrice =
        products.Average(product => product.ListPrice);

    Console.WriteLine("The average list price of all the products is ${0}",
        averageListPrice);
}
```

- Informacja o ilości zamówień dla danego ContactID

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;

    //Can't find field SalesOrderContact
    var query =
        from contact in contacts
        select new
        {
            CustomerID = contact.ContactID,
            OrderCount = contact.SalesOrderHeaders.Count()
        };

    foreach (var contact in query)
    {
        Console.WriteLine("CustomerID = {0} \t OrderCount = {1}",
            contact.CustomerID,
            contact.OrderCount);
    }
}
```

- Liczba kontaktów jako LongInt

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;

    long numberOfContacts = contacts.LongCount();
    Console.WriteLine("There are {0} Contacts", numberOfContacts);
}
```

- Zamówienie o najwyższej/najniższej wartości....

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    Decimal maxTotalDue = orders.Max(w => w.TotalDue);
    Console.WriteLine("The maximum TotalDue is {0}.",
        maxTotalDue);
}
```

- Wartość sprzedaży dla każdego ContactID

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        select new
        {
            Category = g.Key,
            TotalDue = g.Sum(order => order.TotalDue)
        };

    foreach (var order in query)
    {
        Console.WriteLine("ContactID = {0} \t TotalDue sum = {1}",
            order.Category, order.TotalDue);
    }
}
```

- Wszystkie produkty za wyjątkiem trzech pierwszych

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    // LINQ to Entities only supports Skip on ordered collections.
    IQueryable<Product> products = context.Products
        .OrderBy(p => p.ListPrice);

    IQueryable<Product> allButFirst3Products = products.Skip(3);

    Console.WriteLine("All but first 3 products:");
    foreach (Product product in allButFirst3Products)
    {
        Console.WriteLine("Name: {0} \t ID: {1}",
            product.Name,
            product.ProductID);
    }
}
```

- Pierwszych pięć kontaktów

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Contact> first5Contacts = context.Contacts.Take(5);

    Console.WriteLine("First 5 contacts:");
    foreach (Contact contact in first5Contacts)
    {
        Console.WriteLine("Title = {0} \t FirstName = {1} \t Lastname = {2}",
            contact.Title,
            contact.FirstName,
            contact.LastName);
    }
}
```



# Join

- Join pomiędzy tabelami Contact a SalesOrderHeader

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        contacts.Join(
            orders,
            order => order.ContactID,
            contact => contact.ContactID,
            (contact, order) => new
            {
                ContactID = contact.ContactID,
                SalesOrderID = order.SalesOrderID,
                FirstName = contact.FirstName,
                LastName = contact.LastName,
                TotalDue = order.TotalDue
            });

    foreach (var contact_order in query)
    {
        Console.WriteLine("ContactID: {0} "
            + "SalesOrderID: {1} "
            + "FirstName: {2} "
            + "LastName: {3} "
            + "TotalDue: {4}",
            contact_order.ContactID,
            contact_order.SalesOrderID,
            contact_order.FirstName,
            contact_order.LastName,
            contact_order.TotalDue);
    }
}
```

- **GroupJoin (LeftJoin)**
- pomiędzy tabelami Contact i SalesOrderHeader i pobranie ilości zamówień per kontakt

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query = contacts.GroupJoin(orders,
        contact => contact.ContactID,
        order => order.Contact.ContactID,
        (contact, contactGroup) => new
        {
            ContactID = contact.ContactID,
            OrderCount = contactGroup.Count(),
            Orders = contactGroup
        });

    foreach (var group in query)
    {
        Console.WriteLine("ContactID: {0}", group.ContactID);
        Console.WriteLine("Order count: {0}", group.OrderCount);
        foreach (var orderInfo in group.Orders)
        {
            Console.WriteLine("    Sale ID: {0}", orderInfo.SalesOrderID);
        }
        Console.WriteLine("");
    }
}
```

- Wyszukanie pierwszego adresu email zaczynającego się od 'caroline'

```
string name = "caroline";  
using (AdventureWorksEntities context = new AdventureWorksEntities())  
{  
    ObjectSet<Contact> contacts = context.Contacts;  
  
    Contact query = contacts.First(contact =>  
        contact.EmailAddress.StartsWith(name));  
  
    Console.WriteLine("An email address starting with 'caroline': {0}",  
        query.EmailAddress);  
}
```

- Zgrupowanie adresów po kodzie pocztowym i projekcja w postaci typu anonimowego

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query = context.Addresses
        .GroupBy( address => address.PostalCode);

    foreach (IGrouping<string, Address> addressGroup in query)
    {
        Console.WriteLine("Postal Code: {0}", addressGroup.Key);
        foreach (Address address in addressGroup)
        {
            Console.WriteLine("\t" + address.AddressLine1 +
                address.AddressLine2);
        }
    }
}
```

- Zgrupowanie kontaktów po pierwszej literze nazwiska dodatkowo posortowane po tym samym. Projekcja do typu anonimowego

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query = context.Contacts
        .GroupBy(c => c.LastName.Substring(0,1))
        .OrderBy(c => c.Key);

    foreach (IGrouping<string, Contact> group in query)
    {
        Console.WriteLine("Last names that start with the letter '{0}':",
            group.Key);
        foreach (Contact contact in group)
        {
            Console.WriteLine(contact.LastName);
        }
    }
}
```

- Wywołanie [SelectMany](#) do pobrania wszystkich zamówień kontaktu "Zhou". Uwaga na **Contact.SalesOrderHeader** navigation property

```
string lastName = "Zhou";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<SalesOrderHeader> ordersQuery = context.Contacts
        .Where(c => c.LastName == lastName)
        .SelectMany(c => c.SalesOrderHeaders);

    foreach (var order in ordersQuery)
    {
        Console.WriteLine("Order ID: {0}, Order date: {1}, Total Due: {2}",
            order.SalesOrderID, order.OrderDate, order.TotalDue);
    }
}
```

- Wszystkie zamówienia złożone po December 1, 2003 wraz ze szczegółami. Uwaga na **order.SalesOrderDetail** navigation property

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<SalesOrderHeader> query = context.SalesOrderHeaders
        .Where(order => order.OrderDate >= new DateTime(2003, 12, 1));

    Console.WriteLine("Orders that were made after December 1, 2003:");
    foreach (SalesOrderHeader order in query)
    {
        Console.WriteLine("OrderID {0} Order date: {1:d} ",
            order.SalesOrderID, order.OrderDate);
        foreach (SalesOrderDetail orderDetail in order.SalesOrderDetails)
        {
            Console.WriteLine("  Product ID: {0} Unit Price {1}",
                orderDetail.ProductID, orderDetail.UnitPrice);
        }
    }
}
```

# Definiowanie i wywoływanie funkcji



- Klasy [EntityFunctions](#) oraz [SqlFunctions](#) dają dostęp do funkcji lokalnych oraz serwerowych jako części EF
- Proces wywołania funkcji customowych wymaga trzech kroków:
  - Zdefiniowania funkcji w modelu conceptualnym lub zadeklarowaniu jej po stronie serwerowej
  - Dodaniu metody w aplikacji i zmapowaniu jej do funkcji w modelu z wykorzystaniem atrybutu [EdmFunctionAttribute](#)
  - Wywołania funkcji w zapytaniach L2E

- Funkcje zwracające pojedynczą wartość mogą być wywoływane bezośrenio na EF
- Inne funkcje mogą być wywoływane wyłącznie jako zapytania L2E

- Wywołanie funkcji jako części zapytania L2E

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    var products = from p in AWEntities.Products
                    where EntityFunctions.DiffDays(p.SellEndDate, p.SellStartDate) < 365
                    select p;
    foreach (var product in products)
    {
        Console.WriteLine(product.ProductID);
    }
}
```

- Bezpośrednie wywołanie funkcji

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    double? stdDev = EntityFunctions.StandardDeviation(
        from o in AWEntities.SalesOrderHeaders
        select o.SubTotal);

    Console.WriteLine(stdDev);
}
```

- Klasa [SqlFunctions](#) udostępnia funkcje Sql Serwera do wykorzystania w zapytaniach L2E.
- Każde wywołanie metod klasy SqlFunctions powoduje wywołanie odpowiedniej funkcji SqlSerwera
- Metody klasy **SqlFunctions** są specyficzne dla SqlSerwera. Podobne klasy z metodami specyficznymi dla innych serwerów mogą być dostarczane za pośrednictwem innych providerów

# SqlFunction – przykład



- Wywołanie Sql'owej funkcji [CharIndex](#) do znalezienia wszystkich kontaktów, których nazwiska kończą się na "Si"

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    // SqlFunctions.CharIndex is executed in the database.
    var contacts = from c in AWEntities.Contacts
                   where SqlFunctions.CharIndex("Si", c.LastName) == 1
                   select c;

    foreach (var contact in contacts)
    {
        Console.WriteLine(contact.LastName);
    }
}
```

# Wywołanie customowej funkcji SQL



- Żeby wywołać customową funkcję zdefiniowaną w bazie danych należy:
  - Stworzyć funkcję na serwerze (Create Function ...)
  - Zadeklarować funkcję w SSDL w pliku .edmx. Nazwa tej funkcji musi być identyczna z tą zdefiniowaną na serwerze
  - Dodać odpowiednią metodę do aplikacji i dodać mapowanie ([EdmFunctionAttribute](#))
  - Wywołać metodę w zapytaniu L2E

```
USE [School]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE FUNCTION [dbo].[AvgStudentGrade](@studentId INT)
RETURNS DECIMAL(3,2)
AS
    BEGIN
        DECLARE @avg DECIMAL(3,2);
        SELECT @avg = avg(Grade) FROM StudentGrade WHERE StudentID = @studentId;

        RETURN @avg;
    END
```



- Deklarujemy funkcje w pliku .edmx

```
<Function Name="AvgStudentGrade" ReturnType="decimal" Schema="dbo" >  
  <Parameter Name="studentId" Mode="In" Type="int" />  
</Function>
```

- Tworzymy metode w aplikacji i mapujemy do funkcji zadeklarowanej w SSDL'u jak poniżej

```
[EdmFunction("SchoolModel.Store", "AvgStudentGrade")]  
public static decimal? AvgStudentGrade(int studentId)  
{  
    throw new NotSupportedException("Direct calls are not supported.");  
}
```

- Wreszcie, wywołujemy funkcje w zapytaniach L2E

```
using (SchoolEntities context = new SchoolEntities())
{
    var students = from s in context.People
                    where s.EnrollmentDate != null
                    select new
                    {
                        name = s.LastName,
                        avgGrade = AvgStudentGrade(s.PersonID)
                    };

    foreach (var student in students)
    {
        Console.WriteLine("{0}: {1}", student.name, student.avgGrade);
    }
}
```

# Wywołanie funkcji zdefiniowanych w modelu



- Dodać metodę CLR do aplikacji, która zmapuje funkcje zdefiniowaną w modelu
- Wywołać funkcje w zapytaniach L2E

- Zdefiniowanie funkcji w modelu konceptualnym

```
<Function Name="YearsSince" ReturnType="Edm.Int32">  
  <Parameter Name="date" Type="Edm.DateTime" />  
  <DefiningExpression>  
    Year(CurrentDateTime()) - Year(date)  
  </DefiningExpression>  
</Function>
```

- Następnie dodajemy metodę mapującą do aplikacji

```
[EdmFunction("SchoolModel", "YearsSince")]  
public static int YearsSince(DateTime date)  
{  
    throw new NotSupportedException("Direct calls are not supported.");  
}
```

- Wywołujemy funkcje z zapytań L2E

```
using (SchoolEntities context = new SchoolEntities())
{
    // Retrieve instructors hired more than 10 years ago.
    var instructors = from p in context.People
                      where YearsSince((DateTime)p.HireDate) > 10
                      select p;

    foreach (var instructor in instructors)
    {
        Console.WriteLine(instructor.LastName);
    }
}
```

# Deferred vs Immediate Query Execution

# Deferred vs immediate execution



- Po skomponowaniu pytanie konwertowane jest do postaci drzewiastej zrozumiałej dla EF
- Zapytania L2E są wykonywane zawsze kiedy iterujemy po zmiennej zapytania (tzw. deferred execution)
- Mamy możliwość wymuszenia natychmiastowego wykonania zapytania (przydatne np. do cachowania rezultatów) (tzw. immediate query execution)



- W przypadku zapytań zwracających sekwencje/kolekcje wartości zmienna zapytania nigdy nie przechowuje rezultatów – wyłącznie samo zapytanie.
- Wykonanie takiego zapytania jest odraczane do momentu iterowania po zmiennej zapytania (foreach) – tzw. deferred execution
- Oznacza to że mamy możliwość wykonywania zapytania tak często jak chcemy
- Podejście użyteczne kiedy nasza baza danych jest/może być aktualizowana przez inne aplikacje klienckie

# Deferred query execution

- Deferred execution pozwala dodatkowo na:
  - Łączenie wielu zapytań
  - Rozszerzanie zapytań

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Product> productsQuery =
        from p in context.Products
        select p;

    IQueryable<Product> largeProducts = productsQuery.Where(p => p.Size == "L");

    Console.WriteLine("Products of size 'L':");
    foreach (var product in largeProducts)
    {
        Console.WriteLine(product.Name);
    }
}
```

- Jeżeli pytanie zwraca pojedynczą wartość (np. wynik operacji agregujących) jest ono wykonywane od razu (żeby obliczyć wartość zintegrowaną musimy stworzyć sekwencję/kolekcję do jej wyliczenia)
- Natychmiastowe wykonanie może zostać także wymuszone. Jest to przydatne, kiedy chcemy np. cachować wyniki zapytania.
- Aby wymusić natychmiastowe wykonanie zapytania nie zwracającego pojedynczej wartości należy wywołać na zapytaniu jedną z metod **ToList**, **ToDictionary** lub **ToArray**

- Natychmiastowa ewaluacja wyrażenia i konwersja do tablicy

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Product> products = context.Products;

    Product[] prodArray = (
        from product in products
        orderby product.ListPrice descending
        select product).ToArray();

    Console.WriteLine("Every price from highest to lowest:");
    foreach (Product product in prodArray)
    {
        Console.WriteLine(product.ListPrice);
    }
}
```

- Natychmiastowa ewaluacja wyrażenia i konwersja do postaci słownika

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Product> products = context.Products;

    Dictionary<String, Product> scoreRecordsDict = products.
        ToDictionary(record => record.Name);

    Console.WriteLine("Top Tube's ProductID: {0}",
        scoreRecordsDict["Top Tube"].ProductID);
}
```

- Natychmiastowa ewaluacja wyrażenia i konwersja do Listy

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Product> products = context.Products;

    List<Product> query =
        (from product in products
         orderby product.Name
         select product).ToList();

    Console.WriteLine("The product list, ordered by product name:");
    foreach (Product product in query)
    {
        Console.WriteLine(product.Name.ToLower(CultureInfo.InvariantCulture));
    }
}
```

# LazyLoading vs EagerLoading

# LazyLoading vs EagerLoading



- Opóźnione ładowanie (Lazy Loading) jest mechanizmem umożliwiającym pobieranie pewnych danych dopiero w momencie, kiedy są potrzebne. Ważne, że pobieranie następuje w sposób niewidoczny dla programisty.
- Np. faktury i kolekcja sprzedanych produktów.
- W EF wszelkie kolekcje oraz referencje na inne encje domyślnie podlegają opóźnionemu ładowaniu.
- Zatem po odpytaniu bazy danych zostaną zwrócone tylko czyste faktury, bez kolekcji sprzedanych produktów.
- Kolekcje zostaną pobrane dopiero w momencie, gdy użytkownik będzie chciał uzyskać do nich dostęp – np. wywołanie pętli foreach po produktach sprzedanych w ramach danej faktury
- W takim przypadku zostanie wysłane drugie zapytanie SQL do serwera bazy danych w celu zwrócenia listy sprzedanych produktów dla konkretnej faktury.



# LazyLoading vs EagerLoading



- LazyLoading bywa zwykle oczekiwane/pożądane ponieważ rzadko chcemy uzyskać natychmiastowy dostęp do wszystkich referencji w danej encji.
- Rozważmy jednak przykład, w którym mamy okienko przedstawiające zarówno podstawowe dane faktury, jak i jej pozycje
- Potrzebujemy zatem wyświetlić również kolekcję Order\_Detail która domyślnie jest ładowana za pomocą dodatkowego zapytania SQL.
- W tym przypadku powinniśmy skorzystać z tzw. zachłannego ładowania (eager loading), które w jednym zapytaniu do serwera bazodanowego zwróci nam zarówno encję Product, jak i kolekcję Order\_Detail.

- W Entity Framework w celu wykonania zachłannego ładowania należy skorzystać z metody Include podczas wykonywania zapytania, np.:

```
using (var ctx = new NorthwindEntities())  
{  
    customer = ctx.Customer.Include(c => c.Order)  
        .Where(c => c.CompanyName == "CocaCola").FirstOrDefault<Customer>();  
  
    // możemy również nazwę encji przekazać jako string  
    // customer = ctx.Customer.Include("Order")  
        .Where(c => c.CompanyName == "CocaCola").FirstOrDefault<Customer>();  
}
```

# LazyLoading

```
using (var ctx = new NorthwindEntities())
{
    ctx.Configuration.LazyLoadingEnabled = true;

    //ładowanie tylko klientów
    IList<Customer> custList = ctx.Customers.ToList<Customer>();

    Student cust = custList[0];

    //ładowanie zamówień dla klienta
    Order order = cust.Order;
}
```

```
using (var ctx = new NorthwindEntities())
{
    //Loading students only
    IList<Customer> custList = ctx.Customer.ToList<Customer>();

    Customer cust = custList[0];

    //podlicz zamówienia ale nie ładuj
    var OrderCount = ctx.Entry(std).Collection(s => s.Orders).Query().Count<Order>();

    //ładuje zamówienia dla konkretnego klienta
    ctx.Entry(std).Collection(s => s.Orders).Query()
        .Where<Order>(c => c.ShipCity == "Krakow").Load();
}
```

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Product> products = context.Products;

    Product[] prodArray = (
        from product in products
        orderby product.ListPrice descending
        select product).ToArray();

    Console.WriteLine("Every price from highest to lowest:");
    foreach (Product product in prodArray)
    {
        Console.WriteLine(product.ListPrice);
    }
}
```

# Pytania kompilowane

- Jeżeli wiele razy wywołujemy zapytania podobne co do struktury, często możemy zwiększyć wydajność przez kompilację takiego zapytania i później jego wywoływanie z różnymi parametrami.
- Np. często pobieramy klientów z danego miasta przy czym miasto definiowane jest w RunTime'ie przez użytkownika w odpowiednim formularzu
- Klasa [CompiledQuery](#) dostarcza możliwości kompilowania i cachowania zapytań
- Zawiera metodę Compile (w kilku wersjach)
- Zapytanie jest kompilowane raz – podczas pierwszego wykonania. Po skompilowaniu można korzystać z zapytania dostarczając różne wartości parametrów (typy prymitywne) – metoda Invoke()
- Oczywiście niemożliwa jest rekonstrukcja skompilowanego zapytania (taka która zmieniałaby wygenerowany SQL)
- W nowych EF – autokompilacja zapytań włączona domyślnie

- Przykład kompilacji i wykorzystania skompilowanego zapytania przyjmującego parametr typu [Decimal](#) i zwracającego sekwencję zamówień o wartości równej \$200.00

```
static readonly Func<AdventureWorksEntities, Decimal, IQueryable<SalesOrderHeader>> s_compiledQuery2 =
    CompiledQuery.Compile<AdventureWorksEntities, Decimal, IQueryable<SalesOrderHeader>>((
        ctx, total) => from order in ctx.SalesOrderHeaders
                        where order.TotalDue >= total
                        select order);

static void CompiledQuery2()
{
    using (AdventureWorksEntities context = new AdventureWorksEntities())
    {
        Decimal totalDue = 200.00M;

        IQueryable<SalesOrderHeader> orders = s_compiledQuery2.Invoke(context, totalDue);

        foreach (SalesOrderHeader order in orders)
        {
            Console.WriteLine("ID: {0}  Order date: {1} Total due: {2}",
                              order.SalesOrderID,
                              order.OrderDate,
                              order.TotalDue);
        }
    }
}
```



- Kompilacja i wywołanie zapytania przyjmującego parametr **DateTime** i zwracającego sekwencje zamówień złożonych później niż podana data

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var compiledQuery = CompiledQuery.Compile((AdventureWorksEntities ctx, DateTime orderDate) =>
        from order in ctx.SalesOrderHeaders
        where order.OrderDate > orderDate
        select new {order.OrderDate, order.SalesOrderID, order.TotalDue});

    DateTime date = new DateTime(2004, 3, 8);
    var results = compiledQuery.Invoke(context, date);

    foreach (var order in results)
    {
        Console.WriteLine("ID: {0} Order date: {1} Total due: {2}", order.SalesOrderID, order.OrderDate, order.TotalDue);
    }
}
```

- Niektóre części zapytania mogą być wykonywane „na serwerze” a niektóre lokalnie „na kliencie”
- Ewaluacja na kliencie następuje zawsze przed wykonaniem zapytania na serwerze
- W przypadku ewaluacji na kliencie – jej wynik jest wstawiany w miejsce wyrażenia w zapytaniu i następuje wykonanie zapytania na serwerze
- W przypadku części wykonywanych na serwerze – konfiguracja źródła danych nadpisuje zachowanie/semantyke klienta
- Przykłady: operacje na nullach, precyzja operacji numerycznych, porównania na stringach etc.
- Wszelkie wyjątki powstałe podczas wykonywania zapytania na serwerze są przekazywane bezpośrednio do klienta

- Średnia dla typu Decimal z wartości 0.0, 0.0, i 1.0 to 0.3333333333333333333333333333333333333333333 na kliencie i 0.333333 na serwerze
- Porównywanie stringów zależy od konfiguracji serwera (collation, case sensitive etc.)
- Null-equals-null na SQL serwerze zwraca wartość nieokreśloną na CLR true

- Rezultatem wykonania zapytania L2E mogą być:
  - Kolekcja zera lub więcej typowanych obiektów encji
  - Projekcja typów złożonych zdefiniowanych w modelu koncepcyjnym
  - Kolekcja typów zgodnych z CLR wspieranych przez model koncepcyjny
  - Kolekcje inlinowe
  - Typy anonimowe
- Po wykonaniu zapytania EF odpowiada za materializację obiektów wynikowych (z typami zgodnymi z CLR)

- W przypadku kiedy zapytanie zwraca typy prymitywne sam wynik jest niejako „odłączony” od EF
- W przypadku zaś zwrócenia kolekcji typowanych obiektów encji reprezentowanych przez [ObjectQuery](#), obiekty te funkcjonują cały czas w ramachObjectContext't
- A zatem zachowanie (w tym obiektowe) typu śledzenie zmian, relacje child/parent, polimorfizm są realizowane/nadzorowane przez EF

# Zapytania LINQ nie wspierane w L2E



- Pełna lista dostępna na: [http://msdn.microsoft.com/en-us/library/vstudio/bb738550\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/vstudio/bb738550(v=vs.100).aspx)
- W szczególności uwaga na przeładowane operatory zapytań przyjmujące integer jako jeden z argumentów odnoszący się do indeksu pozycji w kolekcjach

# Znane issues

- Problemy z utrzymaniem porządku sortowania
- Problemy ze zgodnością typów
- Obsługa niektórych typów (np UInt)
- Pytania zagnieżdzone (głębiej niż 3 poziom)

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    uint s = UInt32.Parse("48000");

    IQueryable<SalesOrderDetail> query = from sale in context.SalesOrderDetails
   where sale.SalesOrderID == s
   select sale;

    // NotSupportedException exception is thrown here.
    try
    {
        foreach (SalesOrderDetail order in query)
            Console.WriteLine("SalesOrderID: " + order.SalesOrderID);
    }
    catch (NotSupportedException ex)
    {
        Console.WriteLine("Exception: {0}", ex.Message);
    }
}
```



**Dziękuję za uwagę**